



Tema 08

Pregled standardne i STL biblioteke jezika C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

- 1. Uvod**
- 2. Generičke funkcije i klase**
- 3. Standardna biblioteka jezika C++**
- 4. Standardna biblioteka šablonu (STL)**
- 5. Vektorska grafika u jeziku C++**



1. Uvod

- Standardna biblioteka jezika C++
- Standardna biblioteka šablona jezika C++



Standardna biblioteka jezika C++

- Standardna biblioteka predstavlja *skup klasa i funkcija* koje predstavljaju osnovu i sastavni deo ISO standarda jezika C++
 - biblioteka obezbeđuje nekoliko generičkih kontejnera (struktura podataka), funkcije za njihovu upotrebu, funkcijeske objekte, generičke nizove, tokove podataka (interaktivni ulaz-izlaz i rad s fajlovima), kao i neke često potrebne funkcije, npr. kvadratni koren
 - biblioteka uključuje zaglavla *C standarda* (ISO C90) sa sufiksom *.h* koja se više ne koriste
- Mogućnosti Standardne biblioteke deklarisane su u prostoru imena (*namespace*) *std*
 - biblioteka je veoma obimna, tako da se uključivanje celog prostora imena *std* izbegava

Standardna biblioteka šablona jezika C++

- Standardna biblioteka šablona (*Standard Template Library*, STL) uticala je na razvoj mnogih delova Standardne biblioteke jezika C++
 - Alexander Stepanov (od 1979), Meng Lee (od 1992) iz kompanije HP
 - kompanija HP je 1994. godine izvorni kod biblioteke STL dala na besplatno korišćenje
 - biblioteka STL je 1998. godine uključena u standard jezika C++
- *Obe programske biblioteke imaju veliki broj zajedničkih svojstava, iako nijedna ne predstavlja striktni podskup druge*
 - standardna biblioteka šablona je po obimu znatno manja, ali od nje zavise mnogi delovi standardne biblioteke jezika C++

2. Generičke funkcije i klase

- 1.** Upotreba šablonu u jeziku C++
- 2.** Generisanje funkcija i klasa



2.1 Upotreba šablonu u jeziku C++

- Preklapanje operatora omogućava korišćenje istog naziva funkcije za operacije s različitim tipovima podataka, npr.

```
char max (char i, char j) { return i>j ? i : j; }
int max (int i, int j) { return i>j ? i : j; }
float max (float i, float j) { return i>j ? i : j; }
```

Funkcije se tipično razlikuju samo po *tipovima* podataka

- U jeziku C++ moguće je definisanje **šablonu** (*template*) za opis takvih funkcija, gde se *tip* podataka zadaje kao **parametar šablonu** (tzv. *generičke funkcije*)
- Na isti način mogu se opisati i klase, za koje se tipovi nekih polja i parametri metoda mogu definisati naknadno (*generičke klase*)

Definisanje šablonu

- Šablon generičke funkcije ima oblik

```
template <parametar1, parametar2, ...> opis
```

gde *opis* može biti *deklaracija* (prototip) ili *definicija* generičke funkcije ili klase, a *parametri* označavaju tipove ili konstante:

```
class naziv_tipa
```

```
typename naziv_tipa
```

```
naziv_tipa naziv_konstante
```

- Primeri definicije šablonu klase i šablonu novog tipa podataka

```
template <class T> T max (T a, T b) { return a>b ? a : b; }
```

```
template <typename T, int n>
class Vektor {
    T niz [n];
public:
    T& operator[] (int i) { return niz[i]; }
};
```

2.2 Generisanje funkcija i klasa

- Funkcije i klase definisane šablonima generišu se za konkretnе tipove i konstante na mestima njihovog pozivanja
- Pozivanje se vrši izrazima oblika

naziv_funkcije <argument1, argument2, ...>

naziv_Klase <argument1, argument2, ...>

gde *argumenti* mogu biti

naziv_tipa

konstantni_izraz

- Primeri

`int a= max<int>(1,2); // generiše se int max(int,int)`

`char b=max<char>('1','2');// generiše se char max(char,char)`

`Vektor<int,10> vekt1; // niz od 10 elemenata tipa int`

3. Standardna biblioteka jezika C++

- Osnovne komponente Standardne biblioteke
- Implementacija Standardne biblioteke



Osnovne komponente Standardne biblioteke

- Skup višestruko upotrebljivih *komponenti* zasnovanih na *šablonima* koje implementiraju veliki broj opštih struktura podataka i algoritama za njihovo korišćenje
- Osnovni elementi mogu se koristiti pomoću zaglavlja
 - kontejneri
 - opšte funkcije
 - lokalizacija
 - stringovi
 - tokovi
 - niti
 - numerička
 - C biblioteka



Implementacija Standardne biblioteke

- **Apache C++ Standard Library** - besplatna biblioteka otvorenog koda, implementacija međunarodnog standarda za C++ ISO/IEC 14882
 - Apache Software Foundation
- **Microsoft C++ Standard Library** - biblioteka otvorenog koda kompanije Microsoft za Visual C++
 - GitHub, od 2019



4. Standardna biblioteka šablonu (STL)

1. Komponente Standardne biblioteke šablonu
2. Kontejneri
3. Iteratori
4. Algoritmi
5. Funkcijski objekti



4.1 Komponente Standardne biblioteke šablonu

- Osnovne komponente Standardne biblioteka šablonu (templejta) jezika C++ su:
 - kontejneri
 - iteratori
 - algoritmi
 - funkcijski objekti



4.2 Kontejneri

- Kontejneri su objekti (strukture podataka) koji služe za smeštanje i organizovanje drugih objekata
- Npr. šablon `vector<T>` je kontejner za jednodimenzionalna polja, koji po potrebi automatski povećava svoje dimenzije
- Tip `T` može biti osnovni tip ili neka korisnička klasa
 - klase treba pamtiti pomoću pokazivača da se izbegne potencijalni gubitak informacija o izvedenim klasama (*object slicing*)
- Primeri upotrebe (prostor imena `std`)

```
std::vector<std::string> stringovi; // objekti tipa string
std::vector<double> podaci; // vrednosti tipa double
```
- Šabloni kontejnerskih klasa definisani su u zaglavljima:
 - `vector`, `array`, `deque`, `list`, `forward_list`, `map`, `unordered_map`, `set`, `unordered_set` i `bitset`

Zaglavlja kontejnerskih klasa

Zaglavljje	Opis
vector	<code>vector<T></code> je proširivo polje, novi elementi dodaju se na kraj
array	<code>array<T,N></code> je polje fiksnih dimenzija od N elemenata (efikasnije)
deque	<code>deque<T></code> je red koji se može ažurirati s obe strane
list	<code>list<T></code> je dvostruko povezana lista elemenata tipa T
forward_list	<code>forward_list<T></code> je jednostruko povezana lista elemenata tipa T
map	<code>map<K,T></code> je asocijativna lista objekata tipa <i>pair<K,T></i> (<i>K</i> je ključ)
unordered_map	<code>unordered_map<K,T></code> je asocijativna lista objekata tipa <i>pair<K,T></i> za koje nije definisan poredak
set	<code>set<T></code> je kontejner <i>map</i> , gde je element liste istovremeno i ključ
unordered_set	<code>unordered_set<T></code> je kontejner <i>set</i> za čije elemente nije definisan poredak
bitset	<code>bitset<T></code> je klasa koja predstavlja niz bitova, npr. flegova

Alokatori i komparatori

- Većina kontejnerskih struktura automatski se prilagođava broju elemenata koji se u njih smeštaju
- Npr. stvarna forma šablonu `vector<T>` je
`vector<T, Allocator=allocator<T>>`
tako da se može definisati sopstveni alokator (šablon klase)
- Neki kontejneri prepostavljaju poredak svojih elemenata, kao npr. kontejner *map*
`map<K, T, Compare=less<K>, Allocator=allocator<pair<K,T>> >`
- Element strukture *Compare* je funkcionalni objekt koji poređi ključeve tipa K i određuje njihov redosled
- Moguće je definisati sopstveni komparator, npr. "veći od"

Primer: Upotreba šablonu vektora (1/2)

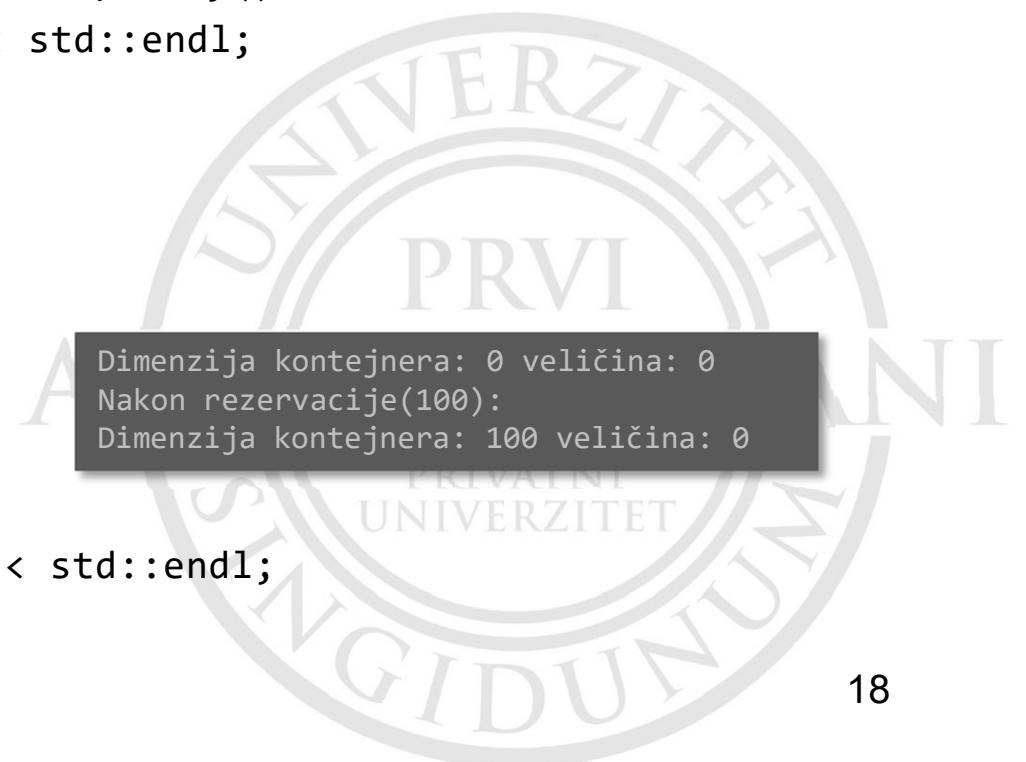
```
#include <iostream>
#include <vector>
using std::vector;

// Šablonska funkcija za prikaz dimenzija i broja elemenata vektora
template<class T>
void listInfo(const vector<T>& v) {
    std::cout << "Dimenzija kontejnera: " << v.capacity()
        << " veličina: " << v.size() << std::endl;
}

int main() {
    // Kreiranje osnovnog vektora
    vector<double> podatak;
    listInfo(podatak);

    // Kreiranje vektora od 100 elemenata
    podatak.reserve(100);
    std::cout << "Nakon rezervacije (100):" << std::endl;
    listInfo(podatak);
```

Dimenzija kontejnera: 0 veličina: 0
Nakon rezervacije(100):
Dimenzija kontejnera: 100 veličina: 0



Primer: Upotreba šablonu vektora (2/2)

```
// Kreiranje i inicijalizacija vektora od 10 elemenata (-1)
vector<int> brojevi(10, -1);

std::cout << "Inicijalna vrednost vektora je: ";
for (auto n : brojevi) std::cout << " " << n; // petlja nad vektorom
std::cout << std::endl << std::endl;

// Provera da li dodavanje elemenata utiče na obim vektora
auto staraDim = brojevi.capacity(); // stari obim
auto novaDim = staraDim;           // novi obim, nakon dodavanja elementa
listInfo(brojevi);
for (int i=0; i<1000; i++) {
    brojevi.push_back(2*i);
    novaDim = brojevi.capacity();
    if (staraDim < novaDim) { // povećanje
        staraDim = novaDim;
        listInfo(brojevi);
    }
}
return 0;
}
```

auto - tip vrednosti određuje inicijalizator, u ovom slučaju tip elementa kontejnera

Dimenzija kontejnera: 0 velicina: 0
Nakon rezervacije (100):
Dimenzija kontejnera: 100 velicina: 0
Inicijalna vrednost vektora je: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Dimenzija kontejnera: 10 velicina: 10
Dimenzija kontejnera: 15 velicina: 11
Dimenzija kontejnera: 22 velicina: 16
Dimenzija kontejnera: 33 velicina: 23
Dimenzija kontejnera: 49 velicina: 34
Dimenzija kontejnera: 73 velicina: 50
Dimenzija kontejnera: 109 velicina: 74
Dimenzija kontejnera: 163 velicina: 110
Dimenzija kontejnera: 244 velicina: 164
Dimenzija kontejnera: 366 velicina: 245
Dimenzija kontejnera: 549 velicina: 367
Dimenzija kontejnera: 823 velicina: 550
Dimenzija kontejnera: 1234 velicina: 824

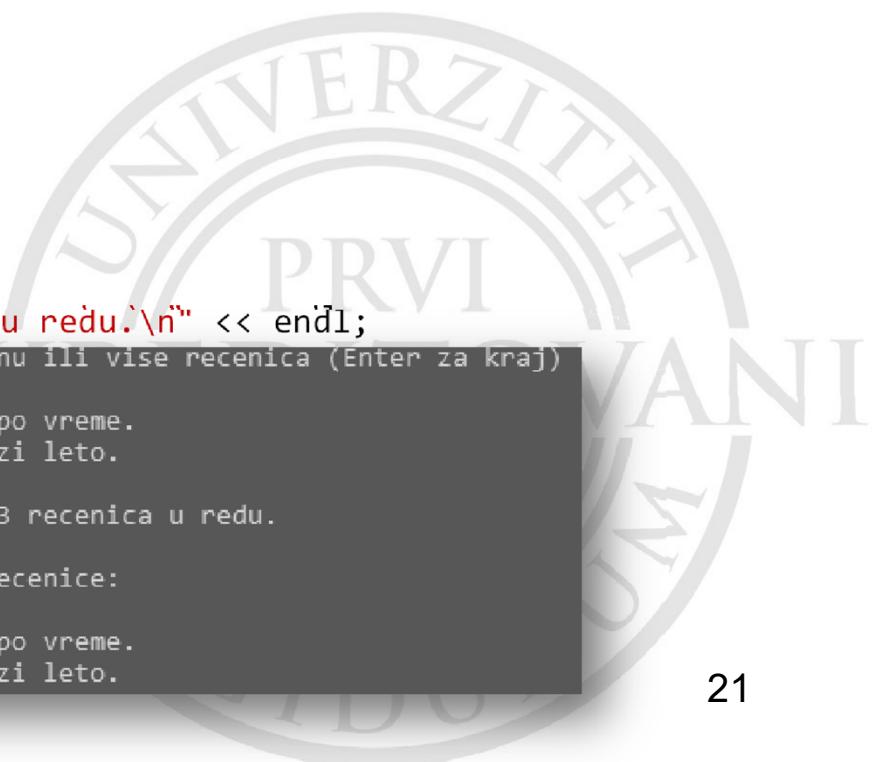
automatsko povećanje dimenzija vektora (capacity) vrši se na način koji zavisi od implementacije; povećanje je na $k \cdot N$ u odnosu na postojeći obim N , gde je k obično 1.5 ili 2

Kontejnerski adapteri

- Kontejnerski adapteri su šabloni klase koji se definišu na osnovu *postojećih* kontejnerskih klasa, obično ograničavanjem njihovih svojstava
- Primeri kontejnerskih adaptera su **red** (*queue*) i **stek** (*stack*), koji se definišu ograničavanjem operacija samo na jednu stranu nekog osnovnog kontejnera
 - **red** se može definisati na osnovu kontejnera **deque<T>** ili **list<T>**
 - **stek** se može definisati na osnovu kontejnera **deque<T>**, **vector<T>** ili **list<T>**
- Adapter **queue** (dostupan preko zaglavlja **<queue>**) dodaje elemente na kraj, a izuzima s početka reda
 - ima metode : **empty()**, **size()**, **front()**, **back()**, **push_back()** i **pop_front()**

Primer: Upotreba šablonu kontejnerskog adaptera *queue*

```
#include <iostream>
#include <queue>
#include <string>
using namespace std;
int main() {
    queue<string> recenice;
    string recenica;
    cout << "Unesite jednu ili vise recenica (Enter za kraj)" << endl;
    while (true) {
        getline(cin, recenica);
        if (recenica.empty())
            break;
        recenice.push(recenica);
    }
    cout << "Ukupno ima " << recenice.size() << " recenica u redu.\n" << endl;
    cout << "Uneli ste recenice:" << endl;
    while (!recenice.empty()) {
        cout << recenice.front() << endl;
        recenice.pop();
    }
    return 0;
}
```



```
Unesite jednu ili vise recenica (Enter za kraj)
Dobar dan.
Danas je lepo vreme.
Uskoro dolazi leto.

Ukupno ima 3 recenica u redu.

Uneli ste recenice:
Dobar dan.
Danas je lepo vreme.
Uskoro dolazi leto.
```

4.3 Iteratori

- **Iteratori** su objekti tipa pokazivača, koji se koriste za pristup objektima kontejnera koji nisu definisani pomoću adaptera
 - iterator za pristup objektima može se dobiti na osnovu kontejnera
 - može se kreirati iterator za ulaz i izlaz objekata ili podataka određenog tipa preko odgovarajućih tokova
 - iteratori imaju iste neke osnovne funkcije, npr. za njihovo poređenje (`==`, `!=` i `=`) i promenu pozicije (inkrement `++` i dekrement `--`)
- Iterator je promenljiva koja pokazuje na podatak u okviru kontejnera
- Svaka kontejnerska klasa ima svoj tip iteratora, ali se iteratori svih kontejnerskih klasa koriste na isti način

Operacija nad iteratorima

- Osnovne operacije nad iteratorom su:
 - prefiksni i postfiksni **inkrement** (**++**) i **dekrement** (**--**) koji pomera iterator na sledeći ili prethodni element kontejnera
 - operatori **==** i **!=**, za ispitivanje da li iteratori pokazuju na isti element
 - operator **indirekcije ***, koji obezbeđuje pristup podatku na koji je iterator pozicioniran (zavisno od konkretne klase kontejnera, pristup može biti za čitanje, pisanje ili i čitanje i pisanje)
- Postoje četiri kategorije iteratora
 - ulazni i izlazni iteratori (*input and output*), samo za čitanje ili upis
 - za obilazak objekata (*forward*), i za čitanje i za upis u jednom smeru
 - bidirekpcioni iteratori, omogućavaju **--iter** i **iter++**
 - iteratori za direktni pristup (*random access*), npr. **iter[n]**, **iter+n**, ...

Funkcije koje vraćaju iteratore

- Mnoge kontejnerske klase imaju funkcije članove koje vraćaju iteratore, koji pokazuju na elemente kontejnera
- Npr. funkcije `std::begin()` i `std::end()` vraćaju iteratore koji pokazuju na *prvi* odnosno *poslednji* element kontejnera - string objekta ili polja koje se prenese kao argument
- Funkcije koje vraćaju vrednost *reverznih* iteratora, koji omogućavaju oblizak sekvenci *unazad* su `std::rbegin()` i `std::rend()`

Upotreba pokazivača

- Osnovni način upotrebe iteratora za obilazak svih elemenata kontejnera je:

```
kontejner<T>::iterator p;  
for (p = kontejner.begin(); p != kontejner.end(); p++)  
    // Obrada elementa kontejnera na poziciji p
```

- Određivanje tipa T promenljive na osnovu tipa dodeljene vrednosti može skratiti kod koji koristi šablove i iteratore
- Umesto pune deklaracije

```
vector<int>::iterator p = v.begin();
```

može se napisati kraće

```
auto p = v.begin();
```

Pametni pokazivači

- Pametni pokazivači (*smart pointers*) su tipovi šabloni slični pokazivačima, koji "pametno" rukuju dinamički alociranim objektima i vrše njihovo automatsko uklanjanje
 - objekt ovog tipa nikad nije potrebno brisati, jer se to vrši automatski, tako da se ne javljaju gubici memorije
- Ovo je moguće jer se upravljanje dinamičkom memorijom vrši na osnovu brojača referenci (*reference counting*). Objekti se brišu kad na njih više ne pokazuje nijedan pokazivač
- Postoje tri tipa šabloni pametnih pokazivača:
 - `unique_ptr<T>` definiše *jedinstveni* objekt određenog tipa
 - `shared_ptr<T>` definiše objekt na koji može pokazivati više *pokazivača*
 - `weak_ptr<T>` sadrži pokazivač povezan s deljivim pokazivačem

4.4 Algoritmi

- **Algoritmi** su šabloni STL funkcija koje vrše operacije nad objektima koje im na raspolaganje stavlja iterator
- Zbog toga algoritmi nemaju informaciju o poreklu objekata, koji mogu biti iz nekog kontejnera ili toka
 - pošto su iteratori poput pokazivača, STL funkcije koje imaju iteratore kao argumente koriste ih kao obične pokazivače
 - algoritmi koriste iteratore za pristup elementima i njihovo smeštanje u niz. Npr. funkcija **sort()** koristi iteratore za pristup elementima radi poređenja i za upis objekata u kontejner u određenom redosledu



Primer: Sortiranje dvostruko povezane liste (1/2)

```
#include <iostream>
#include <deque>
#include <algorithm>           // šablon sort<T>()
#include <numeric>             // šablon accumulate<T>()
#include <functional>          // funktori
using namespace std;

int main() {
    deque<int> podaci;
    // Unos podataka
    cout << "Unesite niz celih brojeva odvojenih razmakom (0 za kraj):" << endl;
    int vrednost = 0;
    while (cin >> vrednost, vrednost != 0)
        podaci.push_front(vrednost);
    // Ispis unesenih podataka
    cout << endl << "Uneli ste brojeve:" << endl;
    for (const auto& n : podaci)
        cout << n << " ";
    cout << endl;
```

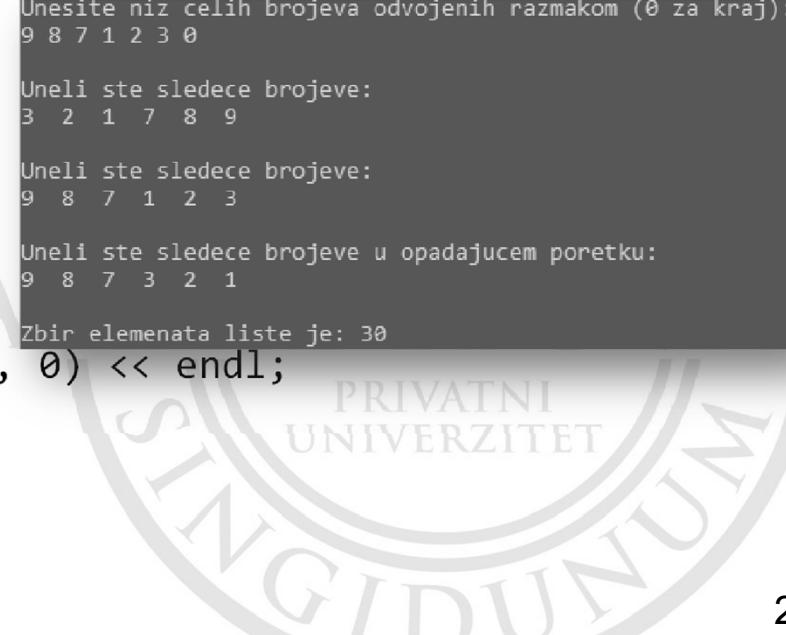


Primer: Sortiranje dvostruko povezane liste (2/2)

```
// Ispis podataka pomoću reverznog iteratora
cout << endl << "Uneli ste, obrnutim redosledom, sledeće brojeve:" << endl;
for (auto riter = crbegin(podaci); riter != crend(podaci); ++riter)
    cout << *riter << " ";
cout << endl;

// Sortiranje liste u opadajućem poretku
cout << endl << "Uneli ste sledeće brojeve u opadajućem poretku:" << endl;
sort(begin(podaci), end(podaci), greater<>()); // sortiranje liste
for (const auto& n : podaci)
    cout << n << " ";
cout << endl;

// Računanje zbira unesenih brojeva
cout << endl << "Zbir elemenata liste je: "
    << accumulate(cbegin(podaci), cend(podaci), 0) << endl;
return 0;
}
```



Unesite niz celih brojeva odvojenih razmakom (0 za kraj):
9 8 7 1 2 3 0
Uneli ste sledeće brojeve:
3 2 1 7 8 9
Uneli ste sledeće brojeve:
9 8 7 1 2 3
Uneli ste sledeće brojeve u opadajućem poretku:
9 8 7 3 2 1
Zbir elemenata liste je: 30

4.5 Funkcijski objekti

- Funkcijski objekti ili *funktori* su objekti tipa klase koji preklapaju operator poziva funkcije, odnosno imaju funkciju-člana **operator()()**
 - implementacija ovog operatora može da vrati rezultat bilo kog tipa
- Npr. korisnička klasa **Linija** s preklopljenim operatorom poziva funkcije može se koristiti kao *funkcija*

```
class Linija {  
    private:  
        double nagib;  
        double y0;  
    public:  
        Linija(double n1_= 1, double y_= 0) : nagib(n1_), y0(y_) {}  
        double operator()(double x) { return y0 + nagib*x; }  
};
```

Primer: Upotreba korisničke klase s preklopljenim operatorom () kao funkcije

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

class Linija {
private:
    double nagib;
    double y0;
public:
    Linija(double nl_= 1, double y_= 0) : nagib(nl_), y0(y_) {}
    double operator()(double x) { return y0 + nagib*x; }
};

int main() {
    Linija f1;
    Linija f2(2.5, 10.0);
    double y1 = f1(12.5); // poziv f1.operator()(12.5) = 0+1*12.5
    double y2 = f2(0.4); // 10.0+2.5*0.4
    cout << "y1=" << y1 << " y2=" << y2 << endl;
}
```

y1=12.5 y2=11

5. Vektorska grafika u jeziku C++

1. Računarska grafika i jezik C++
2. Razvoj grafičkih aplikacija u operativnom sistemu Windows
3. Multiplatformska biblioteka za razvoj grafičkih aplikacija Qt



5.1 Računarska grafika i jezik C++

- Standardna biblioteka jezika C++ i biblioteka šablonu *nemaju podršku za računarsku grafiku i multimedijiske strukture*
 - računarska složenost i hardverska zavisnost
 - razvijeni softver ne bi bio prenosiv
- Biblioteke za vektorsku grafiku, obradu slika/zvuka i razvoj korisničkih interfejsa nisu deo standarda
- Neke poznatije biblioteke za razvoj 2D grafičkih aplikacija i grafičkih korisničkih interfejsa su:
 - Qt multiplatformska, obimna, popularna
 - GTK+ multiplatformska (od GIMP Toolkit)
 - Blend2D biblioteka za vektorsku grafiku razvijena u C++
 - OpenGL *nije objektno orijentisana* (jezik C)

5.2 Razvoj grafičkih aplikacija u operativnom sistemu Windows

- Windows API i biblioteka klase MFC
- Aplikacije operativnog sistema Windows
- Aplikacije koje koriste biblioteku MFC
- Osnovna grafika u grafičkom prozoru
- Prikaz geometrijskih oblika
- Grafika u boji
- Bojenje površina

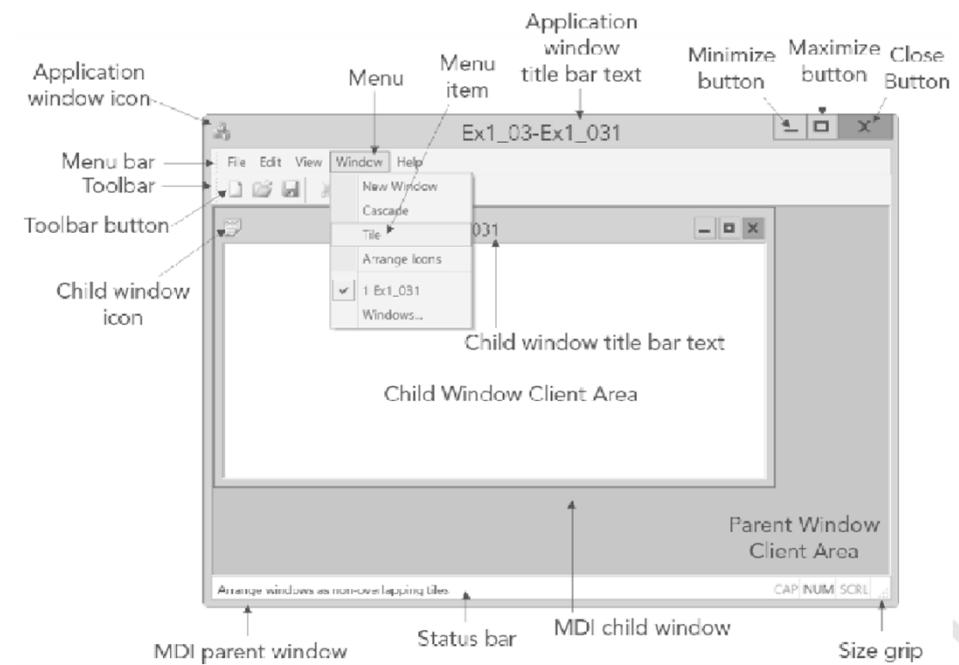


Windows API i biblioteka klase MFC

- Razvoj grafički orijentisanog softvera interfejsa u jeziku C++ za operativni sistem *Windows* može se realizovati na različitim nivoima apstrakcije
 - **direktnim pozivima funkcija operativnog sistema** iz aplikativnog programa preko aplikativnog programskog interfejsa OS Windows ([Windows API](#), Win API, Win 32/64).
To je najteži je i nasporiji način razvoja aplikacija, koji podrazumeva programsko kreiranje svih elemenata interfejsa pozivima funkcija operativnog sistema
 - **upotrebom biblioteke klase Microsoft Foundation Classes (MFC)**, koja obuhvata sve neophodne funkcije Win API. Predstavlja viši nivo programiranja i znatno lakši način realizacije aplikacija

Elementi grafičkog prozora

- Osnovni element grafičkog korisničkog interfejsa je objekt *window* čije su najvažnije komponente:
 - veza s nadređenim i podređenim objektima-prozorima (*parent/child window*)
 - naslov (*title bar*)
 - meni linija
 - paleta alatki (*toolbar*)
 - radna površina (*client area*)
 - statusna linija (*status bar*)



Aplikacije operativnog sistema Windows

- Veliki deo koda *Windows* aplikacija bavi se obradom *događaja* koji su posledica akcija korisnika ili sistemskih događaja
- Operativni sistem beleži događaje u porukama koje stavlja u red čekanja odgovarajućeg programa za obradu događaja
 - program mora imati funkciju namenjenu rukovanju događajima, koja se obično naziva `WndProc()` ili `WindowProc()`
- Minimalni *Windows* program koji koristi Win API sastoji se od dve funkcije:
 - `WinMain()` - funkcija u kojoj počinje izvršavanje osnovnog programa i njegova inicijalizacija
 - `WindowProc()` - funkcija koju Windows poziva radi prenošenja poruka aplikacije

Aplikacije operativnog sistema Windows

- Funkcija *WinMain()* ekvivalent je funkcije *main()* konzolnih programa i ima prototip:

```
int WINAPI WinMain(HINSTANCE hInstance,  
                    HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow);
```

- Osnovna namena funkcije *WinMain()* je:
 - saopštava operativnom sistemu vrstu prozora potrebnu programu
 - kreira i inicijalizuje prozor programa
 - prihvata poruke sistema namenjene programu
- Prototip funkcije *WindowProc()* je:

```
LRESULT CALLBACK WindowProc(HWND hWnd, UINT message,  
                           WPARAM wParam, LPARAM lParam);
```

Aplikacije koje koriste biblioteku MFC

- Biblioteka *Microsoft Foundation Classes* (MFC) je skup klasa koje olakšavaju razvoj *Windows desktop* aplikacija u jeziku Visual C++
- Nazivi svih klasa u biblioteci MFC počinju slovom C, npr. **CDocument** ili **CView**
- Minimalna MFC aplikacija može se kreirati u dva koraka
 1. File/New/Project, izbor tipa projekta *Win32 Project*, a u sledećem dijalogu opcije *Windows Application* i *Empty project*
 2. Iz glavnog menija izbor **Project/Properties**, zatim panela *General*, u kome treba postaviti svojstvo projekta *Use of MFC* na vrednost *Use MFC in a Shared DLL*
- Zatim se kreira novi **cpp** fajl i unese izvorni kod aplikacije

Primer: Minimalna Windows aplikacija koja koristi MFC (1/2)

```
// Elementarni MFC program koji prikazuje prazan prozor
#include <afxwin.h>      // biblioteka klasa

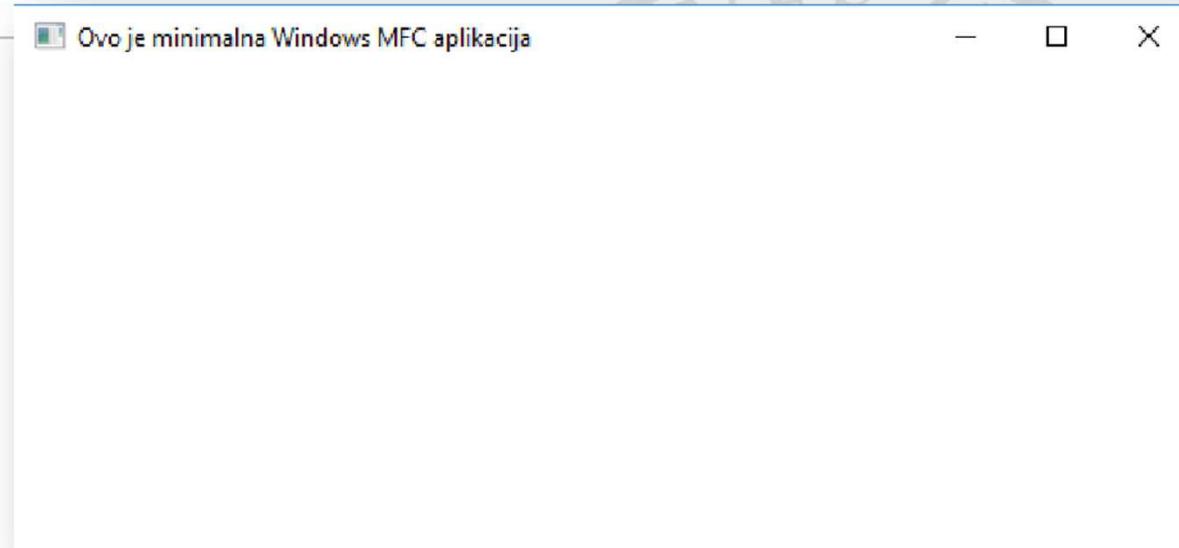
// Definisanje klase aplikacije
class CMojaApp:public CWinApp {
public:
    virtual BOOL InitInstance() override;
};

// Definicija klase Window
class CMojWnd:public CFrameWnd {
public:
    // Konstruktor klase Window
    CMojWnd() {
        Create(nullptr, _T("Ovo je minimalna Windows MFC aplikacija"));
    }
};
```



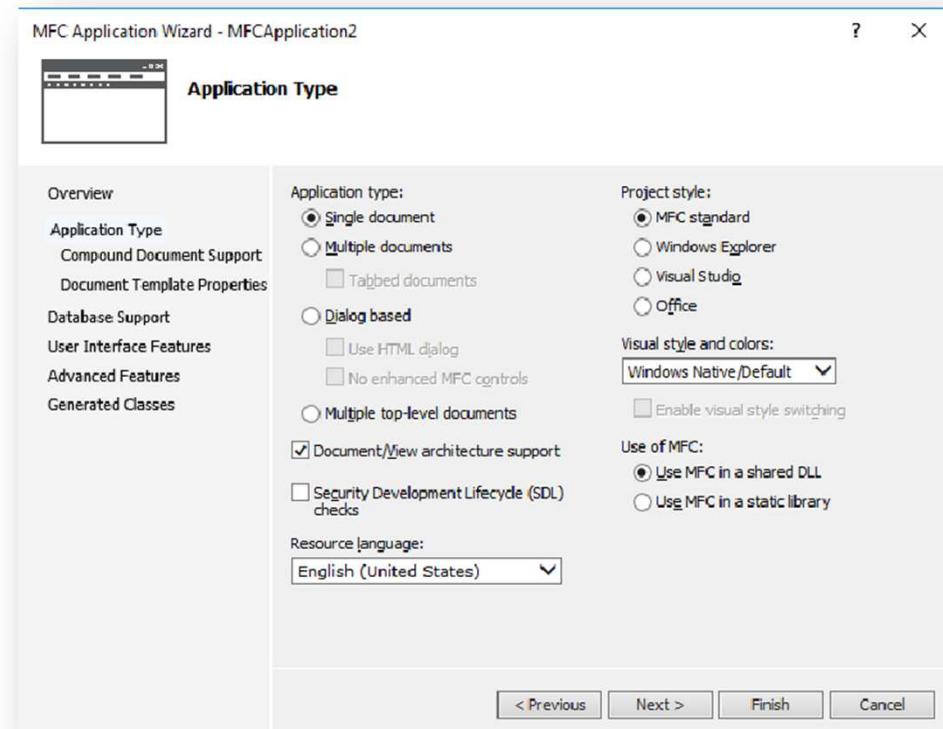
Primer: Minimalna Windows aplikacija koja koristi MFC (2/2)

```
// Funkcija za kreiranje instance glavnog prozora aplikacije
BOOL CMojaApp::InitInstance(void) {
    m_pMainWnd = new CMojWnd;           // Konstruiše objekt window...
    m_pMainWnd -> ShowWindow(m_nCmdShow); // ...i prikazuje prozor
    return TRUE;
}
// Definicija globalnog objekta aplikacije
CMojaApp MojaAplikacija;
```



Kreiranje MFC aplikacije pomoću *Application Wizzarda*

1. Kreira se novi projekt tipa *MFC Application* koristeći *Application Wizzard*
2. Postave se opcije kao na slici i prihvate podrazumevajuće vrednosti do kraja dijaloga (*Finish*)

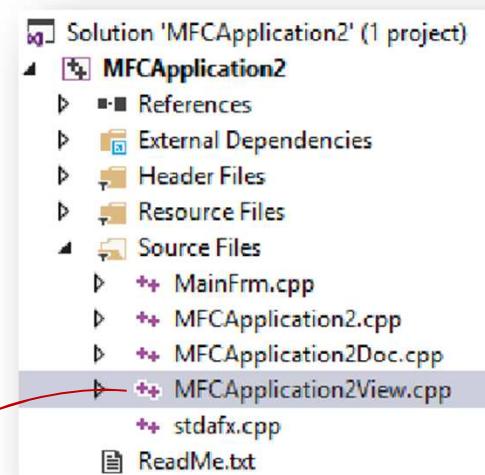


Kreiranje MFC aplikacije pomoću *Application Wizzarda*

- Visual Studio kreira folder MFC aplikacije s programskim kodom raspoređenim u više foldera
- Folder **Source Files** sadrži inicijalni kod MFC aplikacije
- Za prikaz grafike prilagođava se funkcija **OnDraw()** klase **MFCnazivaplikacijeView**

```
void CMFCApplication2View::OnDraw(CDC* /*pDC*/)
{
    CMFCApplication2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: add draw code for native data here
}
```



```
void CMFCApplication2View::OnDraw(CDC* pDC)
{
    CMFCApplication2Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);
    if (!pDoc)
        return;

    // TODO: add draw code for native data here
    pDC->MoveTo(50, 50); // postavljanje tekuće pozicije
    pDC->LineTo(50, 200); // vertikalna linija dole 150 jed.
    pDC->LineTo(150, 200); // horizontalna linija desno 100
    pDC->LineTo(150, 50); // vertikalna linija gore 150 jed.
    pDC->LineTo(50, 50); // horizontalna linija levo 100 jed.
}
```

Osnovna grafika u grafičkom prozoru

- Crtanje po radnoj površini vrši se pomoću generisane funkcije **OnDraw()** klase **CMFCGrafikaView**, npr.

```
void CMFCGrafikaView::OnDraw(CDC* /*pDC*/) {  
    CCMFCGrafikaDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc)  
        return;  
    // Dodavanje koda za crtanje, npr.  
    pDC->MoveTo(50, 50); // postavljanje tekuće pozicije  
}
```

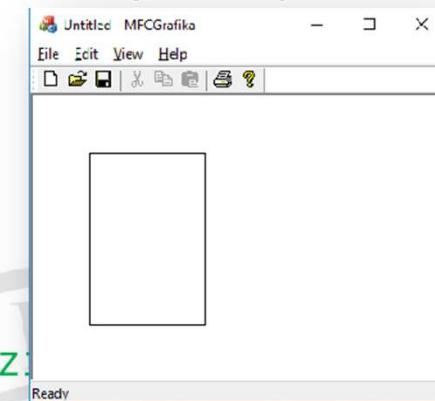
brisati oznake komentara

- Kad god aplikacija dobije poruku **WM_PAINT**, potrebno je ponovo izvršiti crtanje dela ili cele radne povšine prozora, npr. zbog toga što je korisnik promenio veličinu prozora

Prikaz geometrijskih oblika

- Linija se može crtati pozivom metoda `MoveTo()` i `LineTo()`, koja na radnoj površini crta liniju do zadane pozicije, npr.

```
void CMFCGrafikaView::OnDraw(CDC* pDC) {  
    CMFCGrafikaDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc)  
        return;  
    pDC->MoveTo(50,50); // postavljanje tekuće poz.  
    pDC->LineTo(50,200); // vertikalna linija dole 150 jed.  
    pDC->LineTo(150,200); // horizontalna linija desno 100  
    pDC->LineTo(150,50); // vertikalna linija gore 150 jed.  
    pDC->LineTo(50,50); // horizontalna linija levo 100 jed.  
}
```

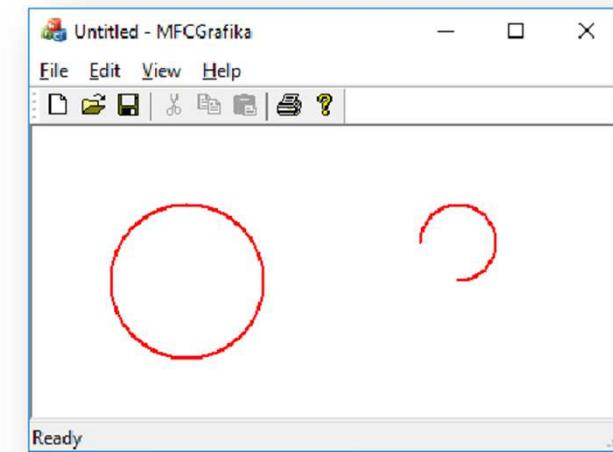


Grafika u boji

- Grafika koristi objekt **pero (pen)** za definisanje linije kojom se crta: boje, debljine i tipa (puna, tačkasta, isprekidana i sl.)
- Objekt se najjednostavnije definiše kreiranjem objekta klase **CPen** pomoću podrazumevajućeg konstruktora klase:
CPen pero;
- Objekt se inicijalizuje zadanim vrednostima svojstava pomoću funkcije **CreatePen()** klase **Cpen**:
`BOOL CreatePen(int stil,int sirina,COLORREF boja);`
- Npr. pero za crtanje pune crvene linije kreira se kao objekt:
`pero.CreatePen(PS_SOLID, 2, RGB(255,0,0));`
 - elipsa i kružnica crtaju se funkcijom **Ellipse()** kojoj se zadaju koordinate dva temena opisanog pravougaonika

Primer: MFC program za crtanje nekoliko geometrijskih figura

```
void CMFCAplication2View::OnDraw(CDC* pDC) {  
    CMFCAplication2Doc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc) return;  
    // Definisanje pera: crvena puna lin. sir. 2px  
    CPen pero;  
    pero.CreatePen(PS_SOLID, 2, RGB(255, 0, 0));  
    CPen* pStaroPero = pDC->SelectObject(&pero); // Promena pera  
    pDC->Ellipse(50,50,150,150); // Crtanje velike kružnice  
    // Definisanje okvira za manju kružnicu (pravougaonika)  
    CRect rect {250,50,300,100};  
    CPoint start {275,100}; // Početna tačka luka  
    CPoint end {250,75}; // Krajnja tačka luka  
    pDC->Arc(&rect,start, end); // Crtanje luka druge kružnice  
    pDC->SelectObject(pStaroPero); // Vraćanje starog pera  
}
```



Bojenje površina

- Površine se boje definisanjem objekta **CBrush** koja sadrži *Windows* četkicu za bojenje površina (puna boja, raster ili neki uzorak)
 - definiše blok dimenzija 8x8 piksela, koji se ponavlja po površini koju treba popuniti
- Za bojenje punom bojom dovoljno je prilikom kreiranja četkice definisati boju, npr.

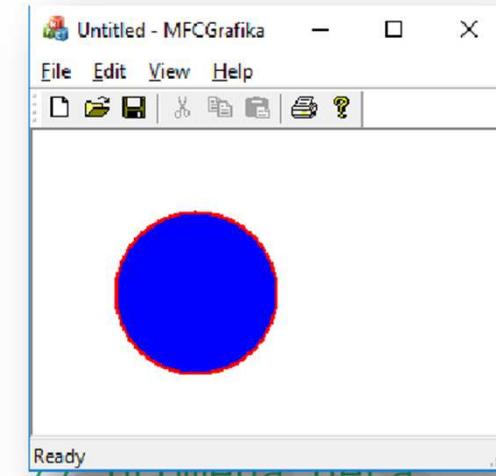
```
CBrush cetkica {RGB(0,0,255)}; // Plava boja popune
```
- Četkica se bira CDC funkcijom **SelectObject()**

```
CBrush* pStara {pDC->SelectObject(&cetakica)};
```

 - stara vrednost četkice se pamti radi mogućnosti restauriranja prethodnog stanja nakon upotrebe četkice

Primer: Obojen krug

```
void CMFCGrafikaView::OnDraw(CDC* pDC) {  
    CMFCGrafikaDoc* pDoc = GetDocument();  
    ASSERT_VALID(pDoc);  
    if (!pDoc) return;  
    // Definisanje pera: crvena puna linija 2px  
    CPen pero;  
    pero.CreatePen(PS_SOLID, 2, RGB(255, 0, 0));  
    CPen* pStaroPero = pDC->SelectObject(&pero);  
    // Definisanje boje popune: plava  
    CBrush cetkica{ RGB(0,0,255) };  
    CBrush* pStara{ pDC->SelectObject(&cetakica) }; // promena četkice  
    pDC->Ellipse(50, 50, 150, 150); // crtanje popunjene kružnice  
    pDC->SelectObject(pStara); // vraćanje prethodne četkice  
    pDC->SelectObject(pStaroPero); // vraćanje prethodnog pera  
}
```



5.3 Multiplatformska biblioteka za razvoj grafičkih aplikacija Qt

- Biblioteka **Qt** je u stvari aplikativni okvir (*framework*) za razvoj multiplatformskih aplikacija s grafičkim interfejsom (GUI)
- Prenosivost multiplatformskih aplikacija koje koriste okvir Qt omogućava se zasebnim prevodenjem za svaku platformu
 - razvija se samo *jedna verzija* aplikacije i prevodi za svaku platformu
- Qt je popularn, ali je njena upotreba znatno složenija nego upotreba većine C++ biblioteka
 - ima niz pomoćih alata, npr. *Qt Creator*, integrисано развојно окружење које pojednostavljuје развој апликација са графичким интерфесима (GUI)
- Nije besplatna, осим за развој softvera отвореног кода
 - download folder <https://www.qt.io/download>

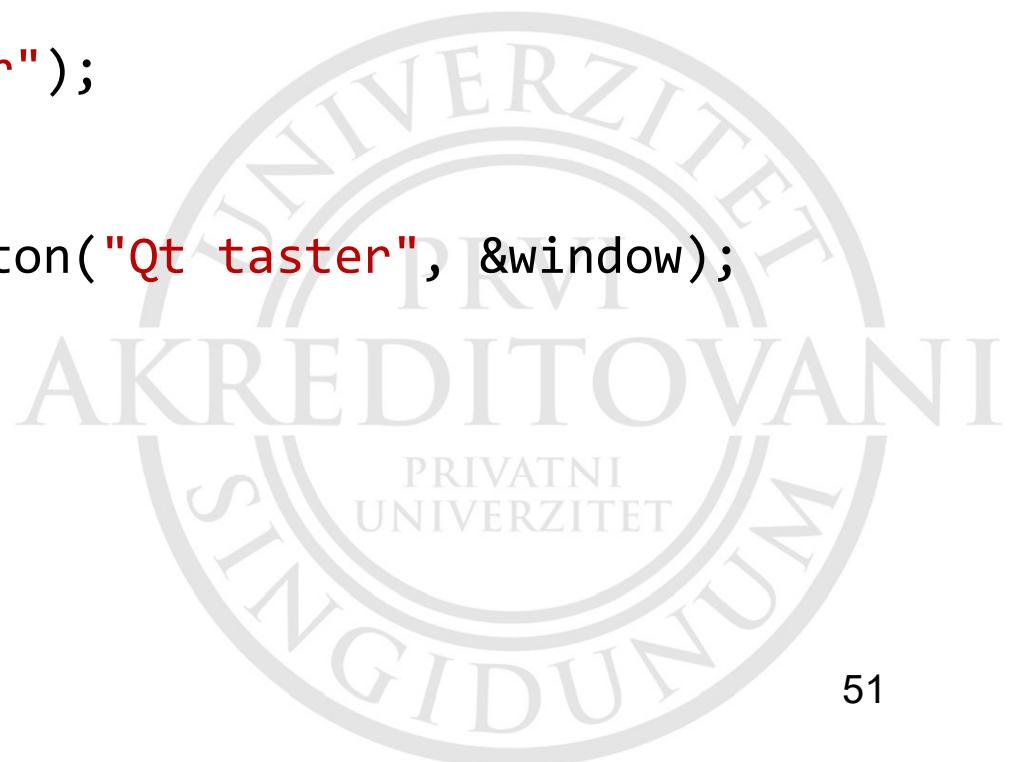
Ilustracija: Jednostavna Qt aplikacija s grafičkim interfejsom (GUI) [8]

```
#include <QtWidgets>
// Primer upotrebe biblioteke Qt
// -- kreiranje grafičkog prozora s jednim tasterom

int main(int argc, char** argv) {
    QWidget window;
    window.resize(120, 100);
    window.setWindowTitle("Qt prozor");
    window.show();

    QPushButton* btn = new QPushButton("Qt taster", &window);

    return app.exec();
}
```



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
8. Grigoryan V., Wu S., *Expert C++: Become a proficient programmer by learning coding best practices with C++17 and C++20*, Packt Publishing, 2020
9. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 09

Kontejneri u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Kontejnerski šabloni
3. Kontejneri sekvenci
4. Asocijativni kontejneri
5. Kontejnerski adapteri
6. Primeri programa



1. Uvod

- Komponente Standardne i STL biblioteke
- Upotreba šablonu iz STL biblioteke
- Kategorije kontejnera



Komponente Standardne i STL biblioteke

- Osnovne komponente **Standardne biblioteke** jezika C++ su sastavni deo ISO standarda, deklarisane u prostoru imena std
 - npr. generički kontejneri (strukture podataka), funkcije za njihovu upotrebu, stringovi, tokovi, niti, numerička i C biblioteka
- Osnovne komponente **STL biblioteke** (Standardne biblioteke šablon/a/templejta) jezika C++ su:
 - kontejneri
 - iteratori
 - algoritmi
 - funkcionalni objekti



Upotreba šablonu iz STL biblioteke

- Upotreba šablonu iz **STL** biblioteke omogućava jednostavnija i kraća programska rešenja brojnih praktičnih problema
- Npr. deo programa koji učitava proizvoljni broj decimalnih vrednosti sa standardnog ulaza i računa njihov prosek:

```
std::vector<double> brojevi;    // zaglavlje <vector>
std::cout << "Unesite brojeve razdvojene prazninama (Ctrl+Z za kraj):\n ";
brojevi.insert(std::begin(brojevi),
               std::istream_iterator<double>(std::cin),
               std::istream_iterator<double>()); // zaglavlje <iterator>
std::cout << "Srednja vrednost = "
         << (std::accumulate(std::begin(brojevi), // zaglavlje <numeric>
                           std::end(brojevi), 0.0)/brojevi.size())
         << std::endl;
```

Kategorije kontejnera

- Kontejneri su objekti (strukture podataka) koji služe za smeštanje i organizovanje drugih objekata
- Osnovne kategorije kontejnera u STL biblioteci su
 - Kontejneri sekvenci (*sequence containers*), u kontinualnoj ili dinamičkoj memoriji (*contiguous storage/list storage*), gde su elementi organizovani u linearne strukture, a pristup elementima se ostvaruje preko funkcije člana ili pomoću iteratora
 - Asocijativni kontejneri, kao što su stabla pretraživanja (*search trees*) i heš tabele (*hash tables*), gde se pristup elementima ostvaruje pomoću ključa ili iteratora
 - Kontejnerski adapteri (*container adapters*) su šabloni klase koji predstavljaju alternativne mehanizme za pristup podacima u kontejnerima sekvenci ili asocijativnim kontejnerima

2. Šabloni kontejnerskih klasa

1. Kontejnerske klase
2. Alokatori
3. Komparatori
4. Zajedničke funkcije kontejnera



2.1 Kontejnerske klase

- Šabloni kontejnerskih klasa definisani su u zaglavljima:
 - `vector`, `array`, `deque`, `list`, `forward_list`, `map`, `unordered_map`, `set`, `unordered_set` i `bitset`
- Npr. šablon `vector<T>` je kontejner za jednodimenzionalna polja, koji po potrebi automatski povećava svoje dimenzije
 - povećanje dimenzija vektora (*capacity*) podrazumeva *kopiranje* postojećih elemenata vektora radi formiranja nove kontinualne memorijske strukture
 - način automatskog povećanja dimenzija zavisi od implementacije; osnovni cilj algoritma povećavanja je da se bitno ne produži prosečno vreme izvršavanja operacija
 - povećanje se vrši na $k \cdot N$ elemenata, gde je N postojeća dimenzija, a faktor k je obično 1.5 ili 2 (Visual C++ koristi $k=1.5$)

Podsetnik: Zaglavlja kontejnerskih klasa

Zaglavlj	Opis
vector	<code>vector<T></code> je proširivo polje, novi elementi dodaju se na kraj
array	<code>array<T,N></code> je polje <i>fiksnih</i> dimenzija od N elemenata (efikasnije)
deque	<code>deque<T></code> je red koji se može ažurirati s obe strane
list	<code>list<T></code> je dvostruko povezana lista elemenata tipa <code>T</code>
forward_list	<code>forward_list<T></code> je jednostruko povezana lista elemenata tipa <code>T</code>
map	<code>map<K,T></code> je asocijativna lista objekata tipa <i>pair<K,T></i> (<i>K</i> je ključ)
unordered_map	<code>unordered_map<K,T></code> je asocijativna lista objekata tipa <i>pair<K,T></i> za koje nije definisan poredak
set	<code>set<T></code> je kontejner <i>map</i> , gde je element liste istovremeno i ključ
unordered_set	<code>unordered_set<T></code> je kontejner <i>set</i> za čije elemente nije definisan poredak
bitset	<code>bitset<T></code> je klasa koja predstavlja niz bitova, npr. flegova

Primer: Promena dimenzija kontejnera (1/2)

```
#include <iostream>
#include <vector>
using std::vector;

// Šablonska funkcija za prikaz dimenzija i broja elemenata vektora
template<class T>
void listInfo(const vector<T>& v) {
    std::cout << "Dimenzija kontejnera: " << v.capacity()
        << " veličina: " << v.size() << std::endl;
}

int main() {
    // Kreiranje osnovnog vektora
    vector<double> podatak;
    listInfo(podatak);

    // Kreiranje vektora od 100 elemenata
    podatak.reserve(100);
    std::cout << "Nakon rezervacije (100):" << std::endl;
    listInfo(podatak);
```

Dimenzija kontejnera: 0 veličina: 0
Nakon rezervacije(100):
Dimenzija kontejnera: 100 veličina: 0

Primer: Promena dimenzija kontejnera (2/2)

```
// Kreiranje i inicijalizacija vektora od 10 elemenata (-1)
vector<int> brojevi(10, -1);
std::cout << "Inicijalna vrednost vektora je: ";
for (auto n : brojevi) std::cout << " " << n; // petlja nad vektorom
std::cout << std::endl << std::endl;

// Provera da li dodavanje elemenata utiče na obim vektora
auto staraDim = brojevi.capacity(); // stari obim
auto novaDim = staraDim;           // novi obim, nakon dodavanja elementa
listInfo(brojevi);
for (int i=0; i<1000; i++) {
    brojevi.push_back(2*i);
    novaDim = brojevi.capacity();
    if (staraDim < novaDim) { // ako se
        staraDim = novaDim;
        listInfo(brojevi);
    }
}
return 0;
}
```

auto - tip vrednosti određuje inicijalizator,
u ovom slučaju tip elementa kontejnera

Dimenzija kontejnera: 0 velicina: 0
Nakon rezervacije (100):
Dimenzija kontejnera: 100 velicina: 0
Inicijalna vrednost vektora je: -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
Dimenzija kontejnera: 10 velicina: 10
Dimenzija kontejnera: 15 velicina: 11
Dimenzija kontejnera: 22 velicina: 16
Dimenzija kontejnera: 33 velicina: 23
Dimenzija kontejnera: 49 velicina: 34
Dimenzija kontejnera: 73 velicina: 50
Dimenzija kontejnera: 109 velicina: 74
Dimenzija kontejnera: 163 velicina: 110
Dimenzija kontejnera: 244 velicina: 164
Dimenzija kontejnera: 366 velicina: 245
Dimenzija kontejnera: 549 velicina: 367
Dimenzija kontejnera: 823 velicina: 550
Dimenzija kontejnera: 1234 velicina: 824

automatsko
povećanje dimenzija
vektora (capacity) na
 $k \cdot N$ elemenata (N je
postojeći obim N , a
faktor $k=1.5$)

Svojstva STL kontejnera

- U STL kontejnerima pamte se *kopije* objekata, osim kad su u pitanju privremeni objekti koji se mogu premeštati
 - na taj način se originalni objekti mogu nezavisno menjati
- Kopiranje složenih objekata može biti neefikasno, pa je bolje u kontejnerima pamtiti *pokazivače* ili koristiti objekte koji se mogu premeštati
- Osim toga, u kontejnere koji služe za pamćenje objekata osnovnih klasa ne treba smeštati objekte izvedenih klasa jer dolazi do gubitka informacija izvedenih klasa (*object slicing*)
- Konstruktori STL kontejnerskih klasa ne prijavljuju izuzetke (*noexcept*)

Svojstva elemenata kontejnera

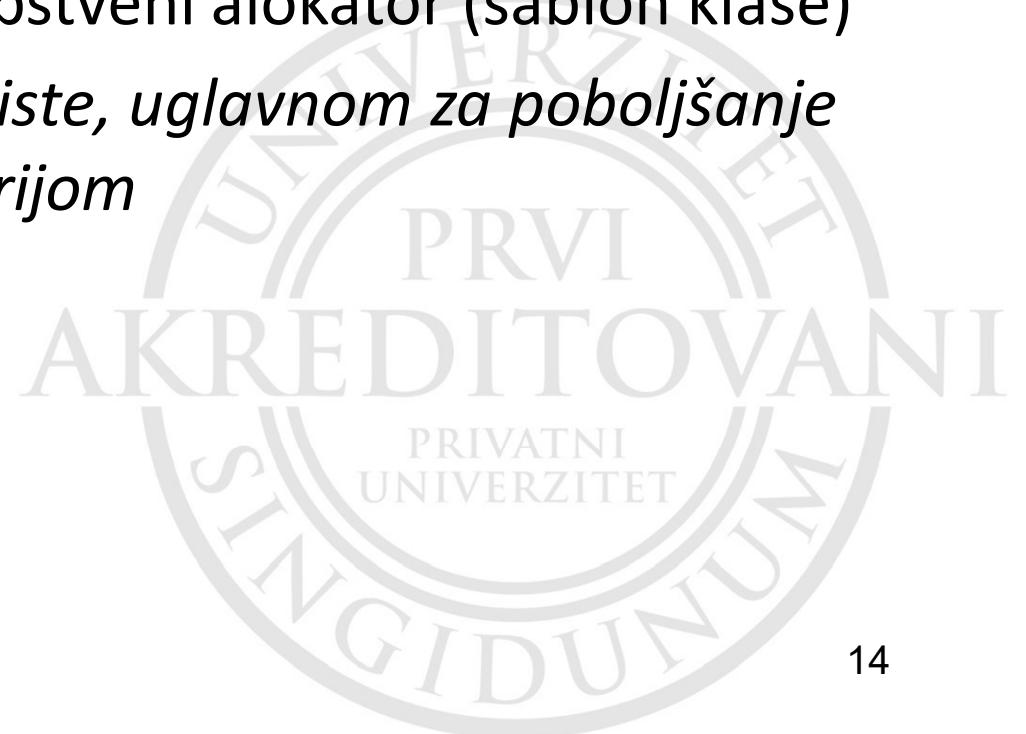
- Element kontejnera tipa T mora imati neka osnovna svojstva, minimalno kao u primeru:

```
class T {
public:
    T();                                // podrazumevajući konstruktor
    T(const T& t);                      // konstruktor kopije
    ~T();                                // destruktur
    T& operator=(const T& t);           // operator dodele
};
```

- Predvodilac najčešće generiše podrazumevajuće elemente za većinu postojećih tipova
- U kontejnerima map i set neophodna je još definicija *funkcije poređenja* elemenata, koja omogućava realizaciju sortiranja

2.2 Alokatori

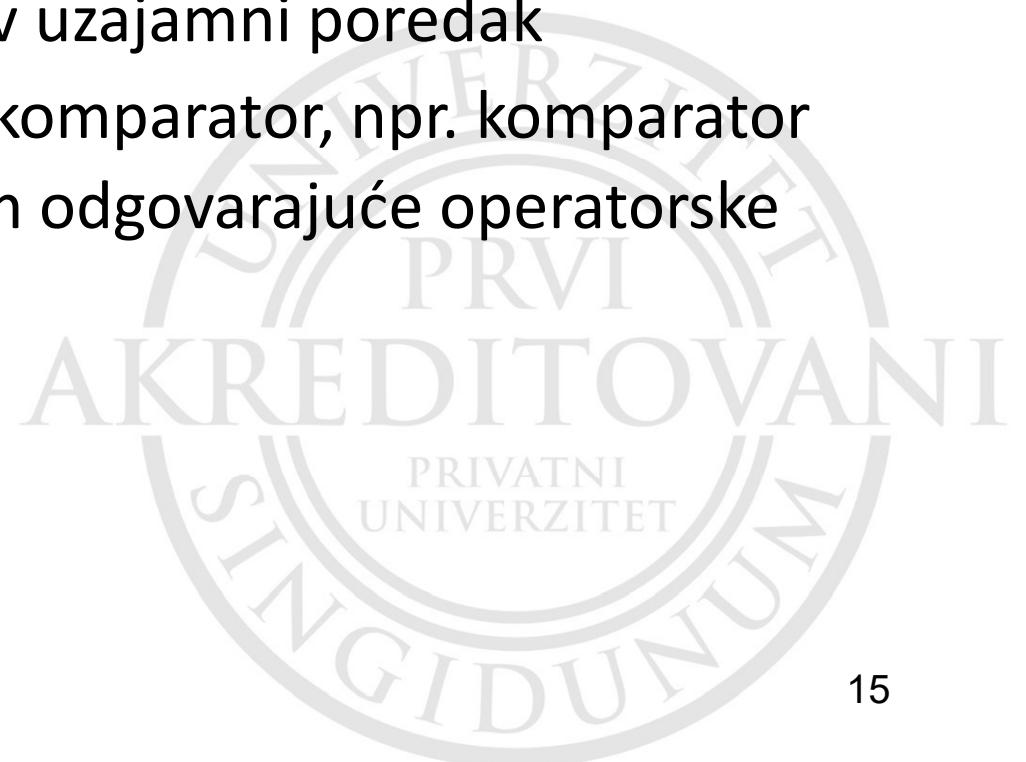
- Većina kontejnerskih struktura automatski se prilagođava broju elemenata koji se u njih smeštaju
- Npr. stvarna forma šablonu `vector<T>` ima još jedan argument `vector<T, Allocator=allocator<T>>`
tako da je moguće definisati sopstveni alokator (šablon klase)
- *Sopstveni alokatori se retko koriste, uglavnom za poboljšanje performansi upravljanja memorijom*



2.3 Komparatori

- Neki kontejneri pretpostavljaju poredak svojih elemenata, kao npr. kontejner *map*

```
map<K, T, Compare=less<K>, Allocator=allocator<pair<K,T>> >
```
- Element strukture *Compare* je funkcionalni objekt koji poredi ključeve tipa *K* i određuje njihov uzajamni poredak
- Moguće je definisati sopstveni komparator, npr. komparator "veći od" dobija se definisanjem odgovarajuće operatorske funkcije **operator>()**



3. Kontejneri sekvenci

1. Svojstva kontejnera sekvenci
2. Zajedničke funkcije kontejnera sekvenci
3. Polja
4. Vektori
5. Liste



3.1 Svojstva kontejnera sekvenci

- Kontinualni kontejneri sekvenci omogućavaju direktni pristup svakom elementu za *konstantno vreme* $O(1)$
- **Polje** je kontejner `std::array<T,N>` i predstavlja standardno polje fiksne veličine N
- **Vektor** je kontejner `std::vector<T>` i predstavlja strukturu čija se veličina menja automatski
 - za kreiranje vektora koristi se dinamička memorija (*heap*)
 - povećanje veličine i brisanje nekog elemenata vektora prouzrokuje *kopiranje* sadržaja u drugi kontinualni blok memorije odgovarajuće veličine, uz istovremeno *oslobađanje* prethodno zauzetog memorijskog bloka

Svojstva kontejnera sekvenci

- Kontejneri tipa **liste** koriste se kada su operacije dodavanja i brisanja elemenata česte
- **Dvostrana lista** (*double-ended queue*), kontejner `std::deque`, predstavlja red čekanja koji čuva elemente u kontinualnim, ali međusobno nezavisnim blokovima dinamičke memorije
 - to omogućava veoma brzo dodavanje elemenata na početak ili kraj, bez njihovog premeštanja
- **Dvostruko povezana lista**, kontejner `std::list`, omogućava obilazak elemenata u dva smera, unapred i unazad
- **Jednostruko povezana lista**, kontejner `std::forward_list`, može se obići za vreme reda $O(n)$, dok je vreme dodavanja i brisanja elemenata konstantno, $O(1)$

3.2 Zajedničke funkcije kontejnера секвеници (1/2)

- Zajedničke funkcije za kontejnerske klase **array**, **vector** i **dequeue** su:

Funkcija	Opis
<code>begin()</code> , <code>end()</code>	vraća iterator begin/end
<code>rbegin()</code> , <code>rend()</code>	vraća reverzni iterator begin/end
<code>cbegin()</code> , <code>cend()</code>	vraća <code>const</code> begin/end iterator (elementi nepromenjivi)
<code>crbegin()</code> , <code>crend()</code>	vraća <code>const</code> reverse begin/end iterator (elementi nepromenjivi)
<code>assign()</code>	prepisuje sadržaj novim skupom elemenata
<code>operator=()</code>	prepisuje element kopijom drugog elementa istog tipa ili liste inicijalizacije
<code>size()</code> , <code>max_size()</code>	vraća aktuelni, odnosno najveći broj elemenata
<code>capacity()</code>	vraća broj alociranih elemenata
<code>empty()</code>	vraća <code>true</code> ako u kontejneru nema elemenata
<code>resize()</code>	menja aktuelni broj elemenata

Zajedničke funkcije kontejnera sekvenci (2/2)

Funkcija	Opis
<code>shrink_to_fit()</code>	smanjuje memoriju potrebnu za aktuelni broj elemenata
<code>front()</code>	vraća referencu na prvi element
<code>back()</code>	vraća referencu na poslednji element
<code>operator[]()</code>	pristupa elementu prema zadanom indeksu
<code>at()</code>	pristupa elementu prema zadanom indeksu uz proveru granica
<code>push_back()</code>	dodaje element na kraj sekvence
<code>insert()</code>	umeće element od zadane pozicije
<code>emplace()</code>	kreira element na zadanoj poziciji
<code>emplace_back()</code>	kreira element na poslednoj poziciji
<code>pop_back()</code>	uklanja element s kraja sekvence
<code>erase()</code>	uklanja jedan ili više elemenata sekvence
<code>clear()</code>	uklanja sve elemente sekvence čija veličina postaje 0
<code>swap()</code>	menja vrednosti dvaju elemenata sekvence
<code>data()</code>	vraća pokazivač na interno polje koje sadrži elemente

3.3 Polja

- Polje `array<T,N>` je najjednostavnija standardna kontejnerska klasa, čiji su parametri šablona tip elemenata `T` i fiksna dimenzija polja `N`
 - pristup elementima polja ima složenost $O(1)$, kao i za ugrađeni tip polja, ali kontejnerska klasa ima dodatne metode linearne složenosti $O(N)$, koje ugrađeni tip polja nema: `begin()`, `end()` i preklopljene operatore `=`, `==` i `<`
- Inicijalizacija polja vrši se pomoću dva para zagrada: spoljašnji par je za objekt `array<T,N>`, a unutrašnji za član `T[N]`
`array<string, 3> {{"jedan", "dva", "tri"}};`
- Ovo omogućava upotrebu polja kao rezultata funkcije, npr.
`return {{"jedan", "dva", "tri"}};`

3.4 Vektori

- Vektor `vector<T>` je kontejner sekvenci koji predstavlja kontinualno polje podataka, čija se veličina može menjati
 - kontinualni raspored podataka omogućava direktni pristup elementima za konstantno vreme $O(1)$
- Dodavanje elemenata zahteva *realokaciju* vektora, kako bi se očuvalo njihov kontinualni raspored u memoriji
 - memorija vektora izuzima se iz dinamičke memorije (*heap*)
 - realokacije se ne vrši za svaki novi element koji se dodaje na kraj; radi efikasnosti, unapred se alocira $k \cdot N$ novih elemenata, gde je N tekuća dimenzija vektora, a k obično 1.5 ili 2
 - broj elemenata koje vektor može da sadrži je njegov kapacitet `capacity()`, a broj vrednosti u vektoru je njegova veličina `size()`, tako da važi $\text{size}() \leq \text{capacity}()$

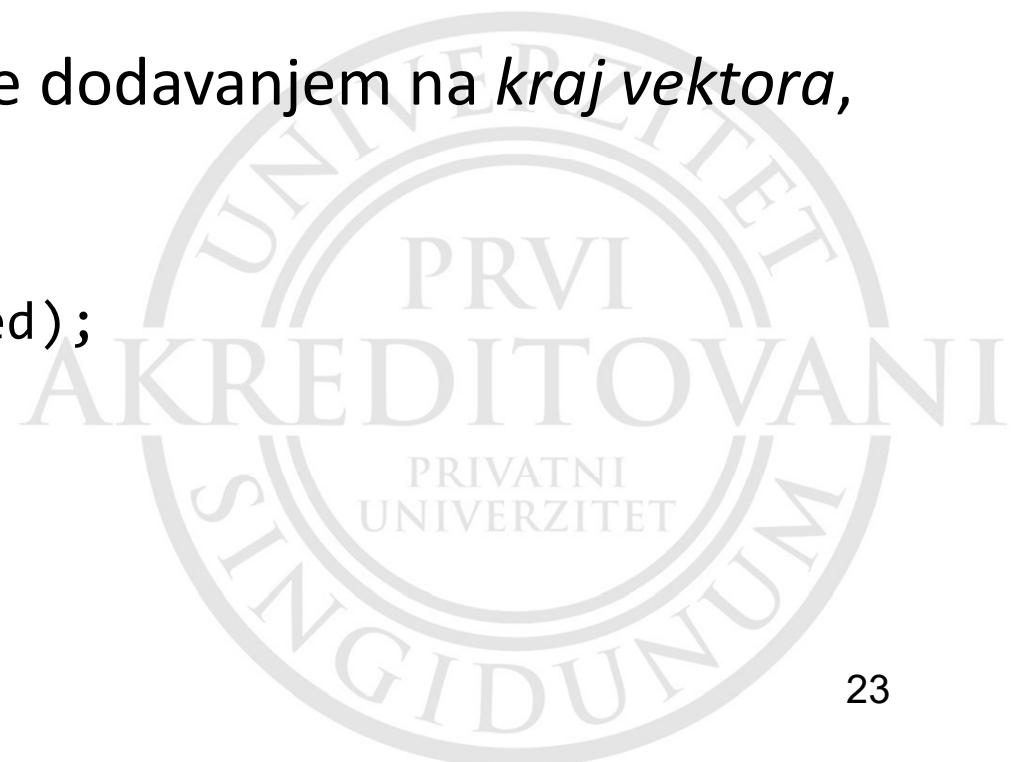
Višedimenzionalni vektori

- Višedimenzionalni vektori definišu se kao vektori vektora; npr. dvodimenzionaln matrica celih brojeva može se definisati kao

```
vector<vector<int>> matrica;
```
- Pristup pojedinačnim elementima se vrši pomoću više indeksa

```
matrica[i][j] = broj;
```
- Proširenje dimenzija se realizuje dodavanjem na *kraj vektora*, a ne pojedinačnog podatka

```
vector<int> red(10)
matrica[i][j] = push_back(red);
```



3.5 Liste

- Kontejner `list<T>` predstavlja dvostruko povezanu listu elemenata tipa `T`, tako da je moguć obilazak liste u oba smera
 - iterator `list<T>::iterator` je *bidirekcion*
- Lista podržava sve operacije koje ima vektor izuzev direktnog pristupa elementima
- Vreme pristupa elementu liste je reda $O(N)$, ali se *dodavanje* na kraj i *izuzimanje* elemenata s kraja liste vrši za konstantno vreme $O(1)$
- Osim dodavanja i uklanjanja elemenata, efikasna operacija je *sortiranje* liste, jer se sastoji samo od međusobne zamene niza pokazivača

4. Asocijativni kontejneri

1. Svojstva asocijativnih kontejnera
2. Skupovi i kolekcije
3. Heš tabele



4.1 Svojstva asocijativnih kontejnera

- Asocijativni kontejneri omogućavaju *sortiranje* i *pretraživanje* elemenata na osnovu relacije porekta i vreme pristupa $O(\log(n))$
 - Skup, kontejner `std::set`, najjednostavija je struktura jedinstvenih, ali uporedivih elemenata
 - Kontejner `std::map` čuva elemente u parovima ključ-vrednost (ključ se koristi za sortiranje i pronalaženje vrednosti elemenata)
 - Kontejneri `std::multiset` i `std::multimap` ne zahtevaju da elementi strukture budu jedinstveni, odnosno međusobno različiti
 - Heš tabele, kontejneri `std::unordered_set` i `std::unordered_map`, omogućavaju pristup (pronalaženje) elemenata za konstantno vreme $O(1)$, s tim da ne zahtevaju jedinstvenost elemenata i ne bave se njihovim poređenjem

4.2 Skupovi i kolekcije

- Skupovi se mogu predstavljati kontejnerskim klasama `set`, `unordered_set` i `multiset`
 - skupovi sadrže samo jedinstvene, međusobno različite elemente, dok se u kolekcijama isti elementi mogu ponavljati
- Pristup elementima ostvaruje se za konstantno vreme $O(1)$
- Između elemenata kontejnera `set` definisana je relacija porekta, tako da su elementi uređeni, dok su kontejneri `unordered_set` skupovi neuređenih elemenata
- Kolekcije se mogu predstaviti kontejnerima `multiset` i `unordered_multiset`, u kojima se isti elementi mogu ponavljati

4.3 Heš tabele

- Heš tabele predstavljaju asocijativne kontejnere kod kojih se pristup elementima realizuje za konstantno vreme $O(1)$
 - elementi heš tabele se pronalaze prema *ključu*, ali ne postoji njihov prirodni poredak, pa se heš tabele ne mogu jednostavno obilaziti u sortiranom redosledu
 - korisnik može da upravlja veličinom heš tabele i da izabere heš funkciju
- Heš tabele se predstavljaju kontejnerskim klasama **`unordered_set`** i **`unordered_map`**
 - u mnogim situacijama mogu se zameniti klasama **`set`** i **`map`**, koje imaju definisan *poredak* elemenata, odnosno relaciju `smaller<`
 - ako je neophodno ponavljanje ključeva, mogu se koristiti kontejnerske klase **`unordered_multiset`** ili **`unordered_multimap`**

Kontejner map

- Kontejner `map<K, T>` služi za smeštanje elemenata tipa `pair<const K,T>` koji sadrži parove elemenata ključ/objekt, gde je ključ tipa `K`, a objekt tipa `T`
 - vrednost ključa mora biti *jedinstvena*, nema duplih ključeva. Poredak objekata definisan je porekom ključeva, koji se međusobno porede funkcijom `less<K>` ili korisnički definisanom funkcijom
- Primer: podaci o starosti osoba u obliku para `<ime,starost>`, gde je *ime* tipa `string`, a *starost* vrednost tipa `size_t`

```
std::map<std::string, size_t> osobe
{{"Ana", 25}, {"Ivana", 33}, {"Bojan", 32}};
```
- Tipični metodi kontejnera `map` i `multimap` su:
`size()`, `empty()`, `max_size()`, `count(k)`, `find(k)`, `lower_bound(k)`,
`upper_bound(k)`, `equal_range(k)` i `swap(k1,k2)`

5. Kontejnerski adapteri

- Kontejnerski adapteri su šabloni klase koji se definišu na osnovu *postojećih* kontejnerskih klasa, najčešće ograničavanjem njihovih svojstava
- Primeri kontejnerskih adaptera su **red** (*queue*), **stek** (*stack*) i **prioriteni red** (*priority queue*), koji se definišu ograničavanjem operacija samo na jednu stranu nekog osnovnog kontejnera
 - **red** se može definisati na osnovu kontejnera `deque<T>` ili `list<T>`
 - **stek** se može definisati na osnovu kontejnera `deque<T>`, `vector<T>` ili `list<T>`



Primeri kontejnerskih adaptera

- Adapter **red** (*queue*, dostupan preko zaglavlja `<queue>`) dodaje elemente na kraj, a izuzima s početka reda
 - ima metode : `empty()`, `size()`, `front()`, `back()`, `push_back()` i `pop_front()`
- Adapter **prioriteni red** (*priority_queue*, dostupan preko zaglavlja `<queue>`)
 - ima metode:
- Adapter **stek** (*stack*, dostupan preko zaglavlja `<stack>`)
 - ima metode:



Primer: Definisanje sopstvenog kontejnerskog adaptera *sortirani vektor*

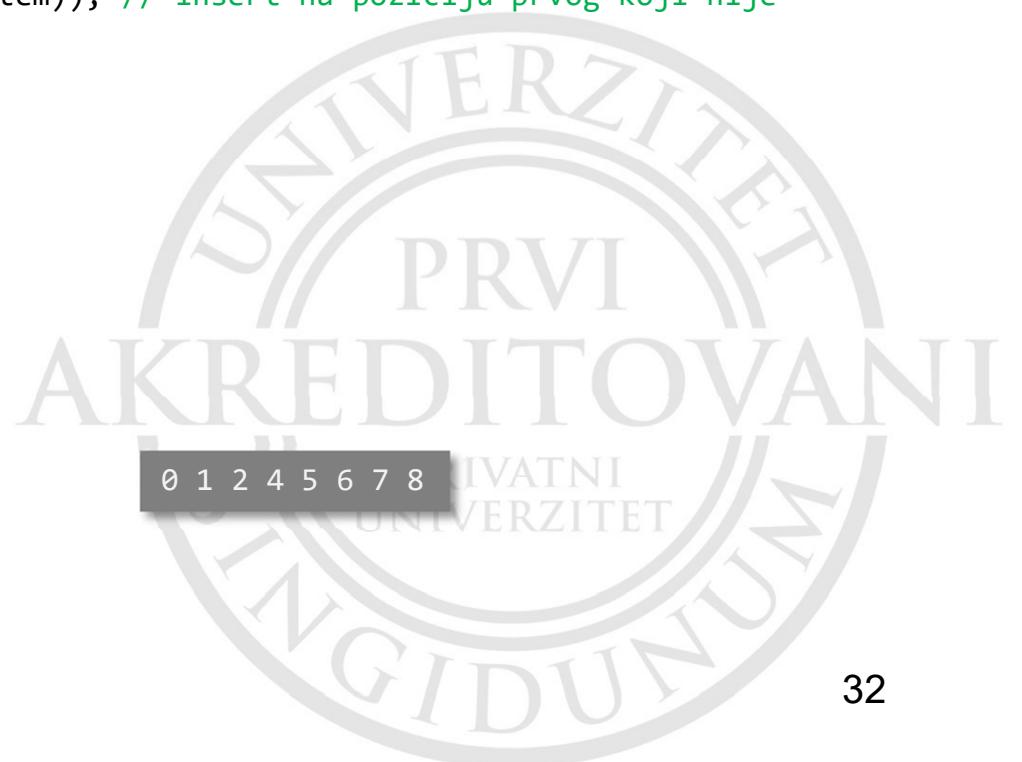
```
// Sopstveni kontejner: sortirani vektor celih brojeva
// - dodavanje elemenata u sortiranom poretku
#include <iostream>
#include <vector>
#include <string>
#include <algorithm> // std::lower_bound, std::sort
#include <iostream>
using namespace std;

void insert_sorted(vector<int> &v, const int &item) {
    const auto insert_pos(lower_bound(begin(v), end(v), item)); // insert na poziciju prvog koji nije
    manji v.insert(insert_pos, item);
}

int main() {
    vector<int> v {6,5,8,2,4};
    sort(begin(v), end(v)); // pocetno sortiranje

    insert_sorted(v, 7);
    insert_sorted(v, 1);
    insert_sorted(v, 0);

    for (const auto &w : v) {
        cout << w << " ";
    }
    cout << '\n'; // 0 1 2 4 5 6 7 8
}
```



0 1 2 4 5 6 7 8

Primer: Definisanje sopstvenog kontejnerskog adaptera *sortirani vektor*

```
// Sopstveni kontejner: sortirani vektor
// - dodavanje elemenata u sortiranom poretku
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iostream>
using namespace std;

template <typename C, typename T>
void insert_sorted(C &v, const T &item) {
    const auto insert_pos(lower_bound(begin(v), end(v), item)); // insert na poziciju prvog koji nije
    manji v.insert(insert_pos, item);
}

int main() {
    vector<int> v {6,5,8,2,4};
    sort(begin(v), end(v)); // pocetno sortiranje
    insert_sorted(v, 7); insert_sorted(v, 1); insert_sorted(v, 0);
    for (const auto &w : v) cout << w << " "; cout << '\n'; // 0 1 2 4 5 6 7 8

    vector<string> v1 {"neke", "slucajne", "reci", "bez", "nekog", "reda"};
    sort(begin(v1), end(v1)); // pocetno sortiranje
    insert_sorted(v1, "zaba"); insert_sorted(v1, "baba");
    for (const auto &w:v1) cout << w << " "; cout << '\n';
}
```

0 1 2 4 5 6 7 8
baba bez neke nekog reci reda slucajne zaba

6. Primeri programa

1. Vektori korisničkih klasa [5]
2. Program za računanje izraza u obrnutoj poljskoj notaciji (RPN) [7]



6.1 Vektori korisničkih klasa

- Definicija klase **Osoba** [Osoba.h]
- Unos podataka o osobama u kontejner **vektor**



Definicija klase Osoba [Osoba.h](1/4)

```
// Klasa definiše osobe po imenima i prezimenima
#pragma once
#include <cstring>
#include <iostream>

class Osoba {
public:
    // Konstruktor, uključuje default
    Osoba(const char* aIme = "Petar", const char* aPrezime = "Petrovic") {
        initImePrezime(aIme, aPrezime);
    }
    // Konstruktor kopije i konstruktor premeštanja
    Osoba(const Osoba& p) {
        initImePrezime(p.ime, p.prezime);
    }
    Osoba(Osoba&& p) {
        ime = p.ime;
        prezime = p.prezime;
        // Brisanje pokazivača na objekt radi prevencije brisanja
        p.ime = nullptr;
        p.prezime = nullptr;
    }
}
```



Definicija klase Osoba [Osoba.h] (2/4)

```
// Destruktor
virtual ~Osoba() {
    delete[] ime;
    delete[] prezime;
}

// Preklopljeni operator <
bool operator<(const Osoba& p) const {
    int result = strcmp(prezime, p.prezime);
    return (result < 0 || result == 0 && strcmp(ime, p.ime) < 0);
}
// Operator dodele
Osoba& operator=(const Osoba& p) {
    // Prevencija dodele oblika p = p
    if (&p != this) {
        delete[] ime;
        delete[] prezime;
        initImePrezime(p.ime, p.prezime);
    }
    return *this;
}
```



Definicija klase Osoba [Osoba.h] (3/4)

```
// Operator dodele i premeštanja
Osoba& operator=(Osoba&& p) {
    // Prevencija dodele p = p
    if (&p != this) {
        // Oslobođanje memorije
        delete[] ime;
        delete[] prezime;
        ime = p.ime;
        prezime = p.prezime;
        p.ime = nullptr;
        p.prezime = nullptr;
    }
    return *this;
}

// Prikaz podataka osobe
void prikaziOsobu() const {
    cout << ime << " " << prezime << endl;
}
```



Definicija klase Osoba [Osoba.h] (4/4)

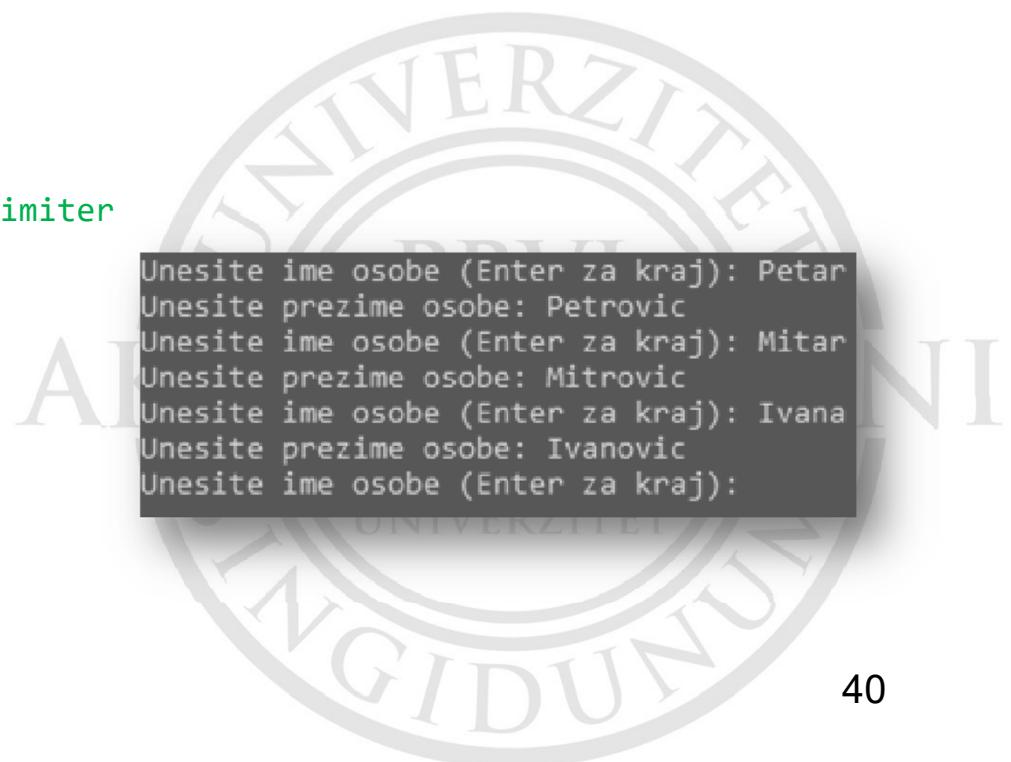
```
private:  
    char* ime{};  
    char* prezime{};  
  
// Pomoćna privatna funkcija za inicijalizaciju imena i prezimena nove osobe  
void initImePrezime(const char* aIme, const char* aPrezime) {  
    size_t length = strlen(aIme) + 1;  
    ime     = new char[length];  
    strcpy_s(ime, length, aIme);  
    length = strlen(aPrezime) + 1;  
    prezime = new char[length];  
    strcpy_s(prezime, length, aPrezime);  
}  
};
```



Unos podataka o osobama u kontejner Vektor(1/2)

```
#include <iostream>
#include <vector>
#include "Osoba.h"
using namespace std;

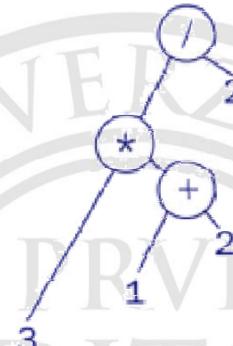
int main() {
    vector<Osoba> ljudi; // Vektor objekata klase Osoba
    const size_t maxduz = 50;
    char ime[maxduz];
    char prezime[maxduz];
    // Unos podataka o ljudima
    while (true) {
        cout << "Unesite ime osobe (Enter za kraj): ";
        cin.getline(ime, maxduz, '\n'); // var,len,delimiter
        if (strlen(ime) == 0)
            break;
        cout << "Unesite prezime osobe: ";
        cin.getline(prezime, maxduz, '\n');
        ljudi.emplace_back(ime, prezime);
    }
}
```



```
Unesite ime osobe (Enter za kraj): Petar
Unesite prezime osobe: Petrovic
Unesite ime osobe (Enter za kraj): Mitar
Unesite prezime osobe: Mitrovic
Unesite ime osobe (Enter za kraj): Ivana
Unesite prezime osobe: Ivanovic
Unesite ime osobe (Enter za kraj):
```

6.2 Program za računanje izraza u obrnutoj poljskoj notaciji (RPN)

- Obrnuta poljska notacija (*Reverse Polish Notation*) je način predstavljanja izraza u fukcionalnom obliku, bez zagrada
 - npr. infiksni aritmetički izraz : $(1 + 2) * 3 / 2$
u funkcionalnom obliku se predstavlja kao: $/ 2 * + 1 2 3$
dok je u obrnutoj (poljskoj) notaciji : $3 2 1 + * 2 /$
 - rezultat računanja izraza je $(1+2)*3/2 = 9/2 = 4.5$
 - izračunavanje (evaluacija) ovakvih izraza vrši se rekurzivnim postupkom, koji se iterativno realizuje pomoću strukture *stek*
(kontejnerski adapter iz STL biblioteke)
- U realizaciji programa koriste se kontejnerski adapter *stack*, kontejner *map* i *lambda* funkcije



Program za računanje izraza u obrnutoj poljskoj notaciji (RPN)

```
#include <iostream>
#include <stack>
#include <iterator>
#include <map>
#include <sstream>
#include <vector>
#include <stdexcept>
#include <cmath>
using namespace std;

template <typename IT>
double evaluate_rpn(IT it, IT end) {
    stack<double> val_stack;
    map<string, double(*)(double, double)> ops{
        { "+", [](double a, double b) { return a + b; } },
        { "-", [](double a, double b) { return a - b; } },
        { "*", [](double a, double b) { return a * b; } },
        { "/", [](double a, double b) { return a / b; } },
        { "^", [](double a, double b) { return pow(a, b); } },
        { "%", [](double a, double b) { return fmod(a, b); } },
    };
}
```

Program za računanje izraza u obrnutoj poljskoj notaciji (RPN)

```
auto pop_stack( [&]()
    { auto r(val_stack.top()); val_stack.pop(); return r; } );
for ( ; it != end; ++it) {
    stringstream ss{ *it };
    double val; ss >> val;
    if (val) {
        val_stack.push(val);
    }
    else {
        const auto r { pop_stack() };
        const auto l { pop_stack() };
        try {
            val_stack.push(ops.at(*it)(l, r));
        }
        catch (const out_of_range &)
            throw invalid_argument(*it);
    }
}
return val_stack.top();
}
```



Program za računanje izraza u obrnutoj poljskoj notaciji (RPN)

```
int main() {
    try {
        cout << evaluate_rpn(istream_iterator<string>{cin}, {}) << '\n';
    }
    catch (const invalid_argument &e) {
        cout << "Neispravan operator " << e.what() << '\n';
    }
    system("pause");
}
```

```
3 2 1 + * 2 /
^Z
4.5
Press any key to continue . . .
```



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 10

Iteratori u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

- 1. Uvod**
- 2. Kontejneri i iteratori**
- 3. Kategorije iteratora**
- 4. Spoljašnje iteratorske funkcije**
- 5. Iteratorski adapteri**
- 6. Upotreba iteratora**



1. Uvod

- Pojam iteratora u jeziku C++
- Ponašanje iteratora
- Osnovni metodi iteratora



Pojam iterатора у језику C++

- Pronalaženje i obrada nizova podataka u starijim verzijama jezika C++ realizovala se korišćenjem *for* petlje
- Pozicija elemenata u nizovima čuvala se u posebnim promenljivima
- Od verzije jezika C++11 uvode se **iteratori**, objekti koji omogućavaju realizaciju obilaska različitih STL kontejnerskih struktura, uz istovremeno pamćenje pozicije elemenata
- Iteratori omogućavaju različite načine obilaska celih struktura ili njihovih delova, element po element

Ponašanje iteratora

- Ponašanje iteratora definiše se pomoću osnovnih operatora:
 - * vraća element na tekućoj poziciji; ako objekt ima članove, pristupa im se pomoću operatora `->`
 - `++` pomera iterator na sledeći element; većina iteratora omogućava prelazak na prethodni element pomoću operatora `--`
 - `==` proverava da li dva iteratora pokazuju na istu poziciju (operator `!=` proverava da li su pozicije različite)
 - `=` dodeljuje vrednost iteratora, u stvari *poziciju* na koju pokazuje
- Iteratori obezbeđuju isti interfejs i različito ponašanje za različite strukture kontejnera na koji se odnose
- Zavisnost od strukture podataka lako se implementira pomoću *šablonu* (templejta)

Osnovni metodi iteratora

- Osnovni metodi iteratora definišu poluotvoreni interval elemenata kontejnera:
 - `begin()` - pozicija početka kontejnera, odnosno prvog elementa kontejnera, ako postoji
 - `end()` - pozicija kraja kontejnera, odnosno pozicija *iza* poslednjeg elemenata kontejnera
- Za svaki tip kontejnera definisana su dva tipa iteratora:
 - `container::iterator` - za obilazak elemenata kontejnera radi čitanja i pisanja (*read/write*)
 - `container::const_iterator` - za obilazak elemenata kontejnera samo radi čitanja (*read only*)

2. Kontejneri i iteratori

- 1. Kontejneri i iteratori**
- 2. Iteratori asocijativnih kontejnera**
- 3. Obilazak kontejnera**



2.1 Kontejneri i iteratori

- Pristup tekućem elementu kontejnera vrši se pomoću iteratora ***pozicija**, a prelazak na sledeći pomoću operatora **++** (prefiksno ili postfiksno)
- Prefiksna upotreba operatora inkrementa (npr. **++pozicija**) može biti efikasnija, jer ne zahteva kreiranje privremenog objekta koji pamti i vraća tekuću poziciju elementa
- Tip iteratora se ne mora eksplisitno definisati, već se može ustanoviti na osnovu prve vrednosti elementa korišćenjem ključne reči **auto**, npr.

```
for (auto poz=lista.begin(); poz!=lista.end(); ++poz) {  
    cout << *poz << ' ';  
}
```

Primer: Prikaz svih elemenata kontejnera

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

int main() {

    list<char> lista; // kontejner liste elemenata tipa char

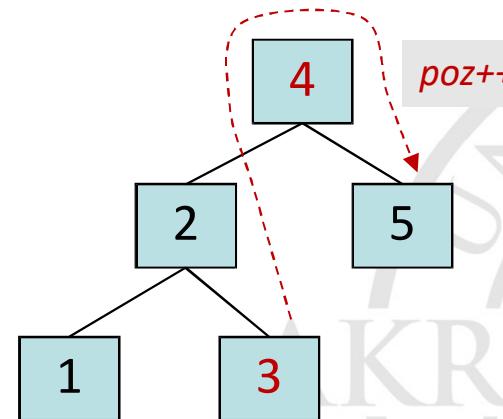
    // Formiranje liste elemenata 'a'-'z'
    for (char c='a'; c<='z'; ++c) {
        lista.push_back(c);
    }
    // Prikaz svih elemenata liste
    list<char>::const_iterator poz; // iterator liste (čitanje)
    for (poz = lista.begin(); poz != lista.end(); ++poz) {
        cout << *poz << ' ';
    }
    cout << endl;
}
```

Rezultat:

a b c d e f g h i j k l m n o p q r s t u v w x y z

2.2 Iteratori asocijativnih kontejnera

- Osim kontejnera sekvenci, iteratori se koriste i za obilazak *asocijativnih* kontejnera, kao što je *skup* (`set`)
- Npr. nakon dodavanja elemenata `1, 2, 3, 4, 5` u skup (u proizvoljnom redosledu) elementi će biti interno sortirani, tako da je uvek desni element veći od levog:



- *Napomena:* kontejner `set` ne dopušta duplike elemenata skupa. Asocijativni kontejner `multiset` može da sadrži više identičnih elemenata

Primer: Obilazak elemenata kontejnera *set* (1)

```
#include <list>
#include <set>
#include <iostream>

int main() {
    // Definisanje skupa elemenata
    typedef std::set<int> SkupCelih; // set<int,greater<int>>
    SkupCelih skup;                // kontejner skupa celobrojnih elemenata

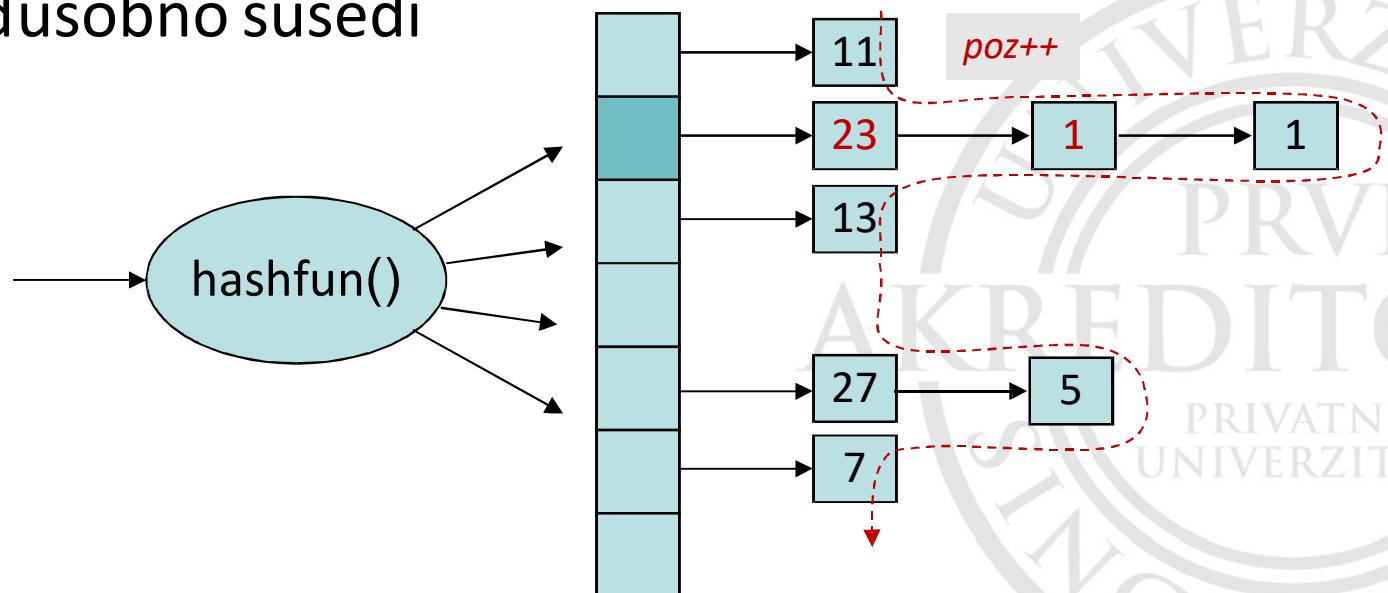
    // Dodavanje 6 elemenata u proizvoljnem redosledu
    skup.insert(3);
    skup.insert(1);
    skup.insert(5);
    skup.insert(4);
    skup.insert(1); // element 1 drugi put
    skup.insert(2);

    // Prikaz svih elemenata skupa
    SkupCelih::const_iterator poz;
    for (poz = skup.begin(); poz != skup.end(); ++poz) {
        std::cout << *poz << ' ';
    }
    std::cout << std::endl;
}
```



Iteratori asocijativnih kontejnera

- Neki kontejneri nemaju definisan poredak elemenata, čak i kad se dodaju prethodno sortirani elementi, npr. asocijativni kontejner `unordered_multiset`
- Dodavanje novog elementa u kontejner menja poredak postojećih elemenata, s tim da su identični elementi uvek međusobno susedi



2.3 Obilazak kontejnera

- Primena iteratora može biti nezavisna od kontejnera, npr.

```
for (auto poz=kont.begin(); poz!=kont.end(); ++poz){  
    ...  
}
```

- Upotreba redosleda elemenata ne važi za sve kontejnere, npr.

```
for (auto poz=kont.begin(); poz<kont.end(); ++poz){  
    ...  
}
```

nije odgovarajući za *liste, skupove i mape*, za koje *redosled* elemenata nije definisan, tako da se ne može se ustanoviti pozicija kraja kontejnera

- greška prevodenja

Obilazak kontejnera

- Obilazak i prikaz elemenata kontejnera može se realizovati sekvencom

```
for (auto elem : kont) {  
    std::cout << elem << ' ';  
}
```

koja je kraća od standardnog načina

```
for (auto poz=kont.begin(); poz!=kont.end(); ++poz){  
    auto elem = *poz;  
    std::cout << elem << ' ';
```

- Svaki kontejner definiše sopstvene tipove iteratora, tako da je posebno zaglavje **<iterator>** potrebno samo za specijalne iteratore, kao što su reverzni i spoljašnji iteratori

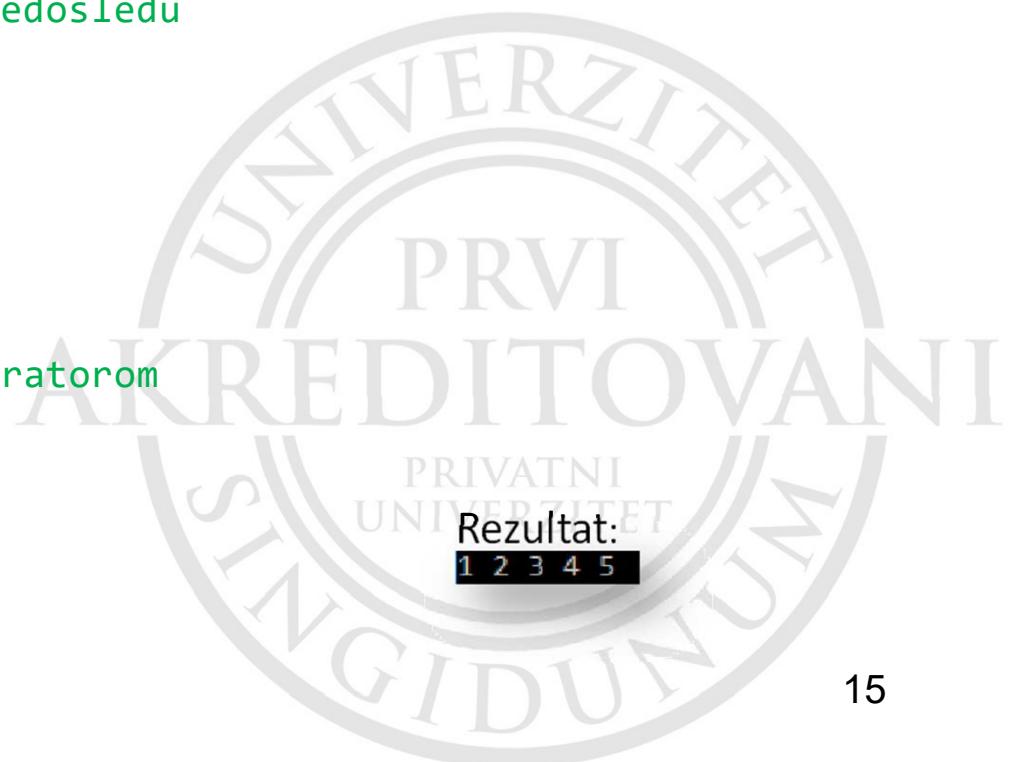
Primer: Obilazak elemenata kontejnera *set* (2)

```
#include <list>
#include <set>
#include <iostream>

int main() {
    // Definisanje skupa elemenata
    typedef std::set<int> SkupCelih; // set<int,greater<int>>
    SkupCelih skup;                // kontejner skupa celobrojnih elemenata

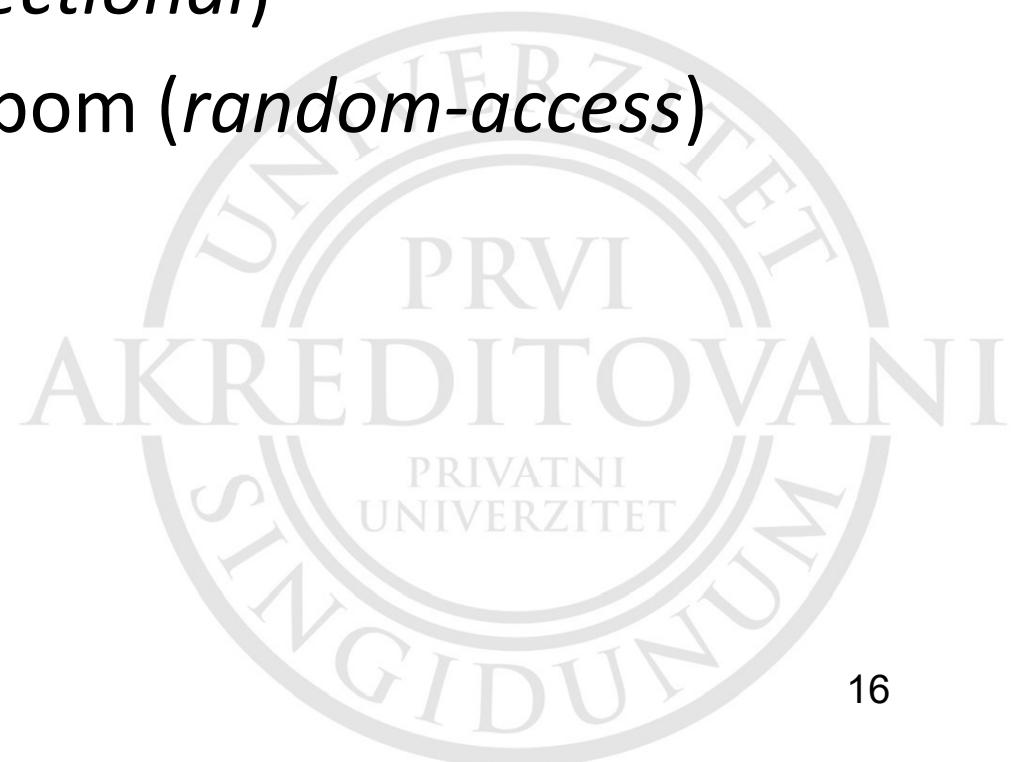
    // Dodavanje 6 elemenata u proizvoljnem redosledu
    skup.insert(3);
    skup.insert(1);
    skup.insert(5);
    skup.insert(4);
    skup.insert(1); // element 1 drugi put
    skup.insert(2);

    // Prikaz svih elemenata skupa s auto iteratorom
    for (auto element : skup) {
        std::cout << element << ' ';
    }
    std::cout << std::endl;
}
```



3. Kategorije iteratora

1. Izlazni iteratori (*output*)
2. Ulazni iteratori (*input*)
3. Jednosmerni iteratori (*forward*)
4. Bidirekpcioni iteratori (*bidirectional*)
5. Iteratori s direktnim pristupom (*random-access*)



Vrste iteratora

- Iteratori imaju neka opšta svojstva, kao i svojstva koja zavise od *tipa kontejnera* i mogu biti:
 1. **Jednosmerni (forward)** - omogućavaju iteraciju samo u jednom smeru pomoću operatora inkrementa
 - za klase kontejnera: `forward_list`, `unordered_set`, `unordered_multiset`, `unordered_map` i `unordered_multimap`
 2. **Dvosmerni (bidirectional)** - omogućavaju iteraciju u oba smera, unapred i unazad, koristeći operatore inkrementa i dekrementa
 - za klase kontejnera `list`, `set`, `multiset`, `map` i `multimap`
 3. **Iteratori s direktnim pristupom (random-access)** - imaju sva svojstva bidirekcionih iteratora, ali omogućavaju i direktni pristup elementima
 - za klase kontejnera `vector`, `deque`, `array` i `string`
 4. **Izlazni (output)** - jednosmerni iteratori za izlazne klase tokova
 5. **Ulazni (input)** - jednosmerni za ulazne klase tokova

3.1 Izlazni iteratori (*output*)

- Omogućavaju samo upis pojedinačnih elemenata u jednom smeru, npr. za upis na standardni izlaz
- Ne može se dva puta dodeliti vrednost bez pomeranja iteratora na sledeću poziciju, odnosno upis se vrši kao

```
while (...) {  
    *pozicija = ... ; // dodata vrednosti  
    ++pozicija;        // pomeranje, za sledeću dodelu  
}
```

- Nijedan iterator klase `const_iterators` nije izlazni i ne omogućava upis

Expression	Effect
<code>*iter = val</code>	Writes <code>val</code> to where the iterator refers
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward (returns old position)
<code>TYPE(iter)</code>	Copies iterator (copy constructor)

3.2 Ulazni iteratori (*input*)

- Omogućavaju samo čitanje pojedinačnih elemenata u jednom smeru, npr. sa standardnog ulaza
 - ne može se dva puta pročitati vrednost bez pomeranja iteratora na sledeću poziciju
- Operatori `==` i `!=` proveravaju da li je vrednost iteratora jednaka ili različita od pozicije iza poslednjeg elementa, tj.

```
InputIterator poz, end;  
while (poz != end) {  
    ... // pristup radi  
        // čitanja pomoću  
        // *poz  
    ++poz;  
}
```

Expression	Effect
<code>*iter</code>	Provides read access to the actual element
<code>iter ->member</code>	Provides read access to a member of the actual element
<code>++iter</code>	Steps forward (returns new position)
<code>iter++</code>	Steps forward
<code>iter1 == iter2</code>	Returns whether two iterators are equal
<code>iter1 != iter2</code>	Returns whether two iterators are not equal
<code>TYPE(iter)</code>	Copies iterator (copy constructor)

3.3 Jednosmerni iteratori (*forward*)

- Iteratori s dodatnim osobinama u odnosu na ulazne; dva jednaka jednosmerna iteratora i nakon inkrementiranja pokazuju na istu vrednost, tj.

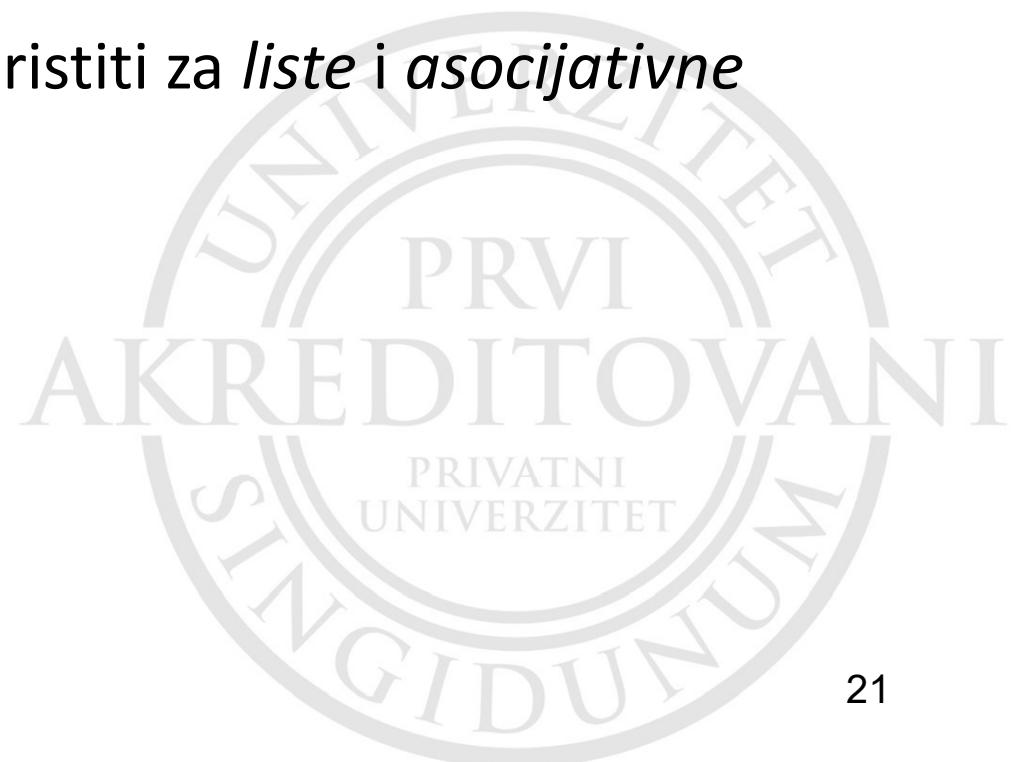
```
ForwardIterator poz1, poz2;
poz1 = poz2 = begin; // pokazuju na isti element
if (poz1 != end) {
    ++poz1; // poz1 je za jedan element ispred poz2
    while (poz1 != end) {
        if (*poz1 == *poz2) {
            ... // obrada jednakih na
                 //susednim pozicijama
            ++poz1;
            ++poz2;
        }
    }
}
```

Expression	Effect
*iter	Provides access to the actual element
iter->member	Provides access to a member of the actual element
++iter	Steps forward (returns new position)
iter++	Steps forward (returns old position)
iter1 == iter2	Returns whether two iterators are equal
iter1 != iter2	Returns whether two iterators are not equal
TYPE()	Creates iterator (default constructor)
TYPE(iter)	Copies iterator (copy constructor)
iter1 = iter2	Assigns an iterator

- Jednosmerni iteratori se mogu koristiti za `forward_list` i kontejnere bez poretku elemenata

3.4 Bidirekpcioni iteratori (*bidirectional*)

- Jednosmerni iteratori s dodatnim svojstvom da omogućavaju iteriranje elemenata *u suprotnom smeru* pomoću operatora dekrementa, i to
 - iter pomera se jedan korak unazad i vraća *novu* poziciju
 - iter-- pomera se jedan korak unazad i vraća *staru* poziciju
- Dvosmerni iteratori mogu se koristiti za *liste* i *asocijativne kontejnere*



3.5 Iteratori s direktnim pristupom (random-access)

- Imaju sva svojstva dvosmernih iteratora, uz mogućnost *drektnog* pristupa
- Podržavaju poređenje i iteratorsku aritmetiku
- Koriste se za kontejnere s direktnim pristupom ([array](#), [vector](#), [deque](#)), stringove i standardna fiksna C-polja

Expression	Effect
$iter[n]$	Provides access to the element that has index n
$iter += n$	Steps n elements forward (or backward, if n is negative)
$iter -= n$	Steps n elements backward (or forward, if n is negative)
$iter + n$	Returns the iterator of the n th next element
$n + iter$	Returns the iterator of the n th next element
$iter - n$	Returns the iterator of the n th previous element
$iter1 - iter2$	Returns the distance between $iter1$ and $iter2$
$iter1 < iter2$	Returns whether $iter1$ is before $iter2$
$iter1 > iter2$	Returns whether $iter1$ is after $iter2$
$iter1 \leq iter2$	Returns whether $iter1$ is not after $iter2$
$iter1 \geq iter2$	Returns whether $iter1$ is not before $iter2$



Primer: Upotreba iteratora s direktnim pristupom (1/2)

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

int main() {
    vector<int> kont;
    // Umetanje elemenata -3 do 9
    for (int i=-3; i <= 9; ++i) {
        kont.push_back (i);
    }
    // Prikaz broja elemenata na osnovu distance između početka i kraja
    // - koristi operator '-' za iteratore
    cout << "Broj elemenata/distanca: " << kont.end() - kont.begin() << endl;
    // Prikaz svih elemenata
    // - koristi operator '<' umesto '!='
    vector<int>::iterator poz;
    for (poz= kont.begin(); poz < kont.end(); ++poz) {
        cout << *poz << ' ';
    }
    cout << endl;
```

Rezultat:
Broj elemenata/distanca: 13
-3 -2 -1 0 1 2 3 4 5 6 7 8 9

Primer: Upotreba iteratora s direktnim pristupom (2/2)

```
// Prikaz svih elemenata
// - koristi operator [] umesto operatora *
for (int i=0; i < kont.size(); ++i) {
    cout << kont.begin()[i] << ' ';
}
cout << endl;
// Prikaz svakog drugog elementa
// - koristi operator +=
for (poz = kont.begin(); poz < kont.end()-1; poz += 2) {
    cout << *poz << ' ';
}
cout << endl;
}
```

Rezultat:

```
Broj elemenata/distanca: 13
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -2 -1 0 1 2 3 4 5 6 7 8 9
-3 -1 1 3 5 7
```

4. Spoljašnje iteratorske funkcije

1. Pomeri (*advance*)
2. Sledeći (*next*) i prethodni (*prev*)
3. Razmak (*distance*)
4. Zamena (*iter_swap*)



4.1 Pomeri (*advance*)

- Standardna biblioteka obezbeđuje nekoliko funkcija za rad s iteratorima
 - advance(), next(), prev(), distance(), iter_swap()
- Funkcija advance() menja poziciju zadanog iteratora unapred i unazad za zadani broj elemenata:

```
#include <iterator>
void advance (InputIterator& poz, Dist n)
```



Primer: Upotreba spoljašnje iteratorske funkcije advance()

```
#include <iterator>
#include <iostream>
#include <list>
#include <algorithm>
using namespace std;

int main() {
    list<int> kont;
    // Umetanje elemenata 1 do 9
    for (int i=1; i<=9; ++i) {
        kont.push_back(i);
    }
    list<int>::iterator poz = kont.begin();

    // Prikaz tekućeg elementa
    cout << *poz << endl;

    // Premeštanje za tri elementa unapred
    advance(poz, 3);

    // Prikaz tekućeg elementa
    cout << *poz << endl;
}

// Premeštanje za jedan element unazad
advance(poz, -1);

// Prikaz tekućeg elementa
cout << *poz << endl;
```

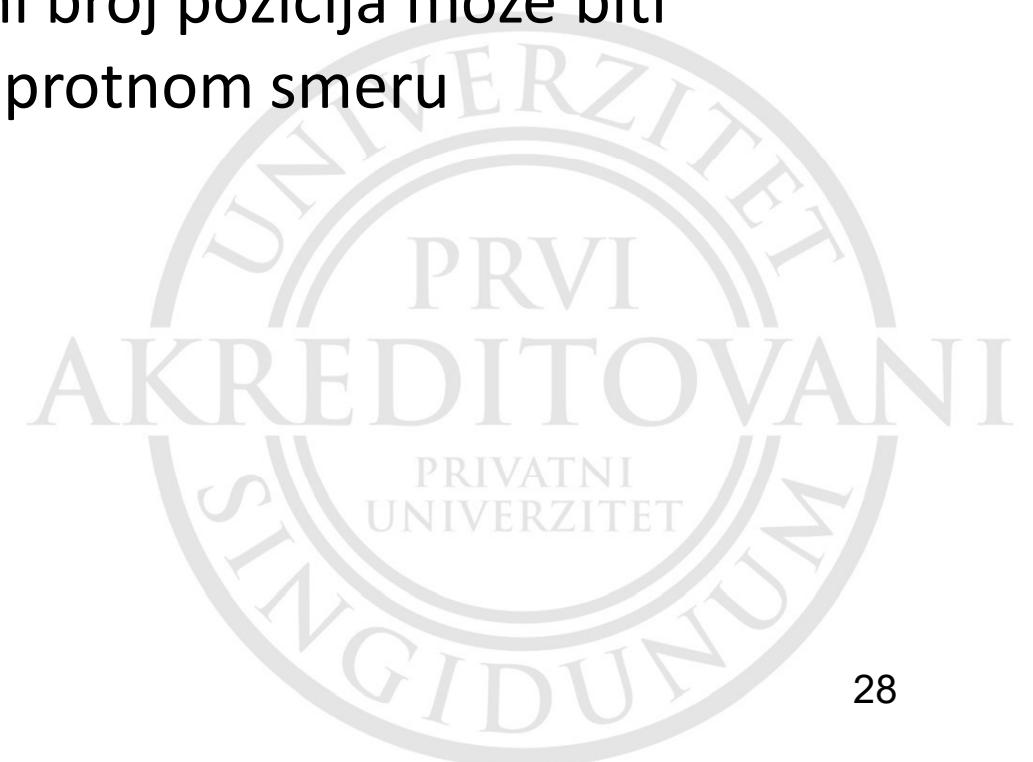
}

Rezultat:

```
1
4
3
```

4.2 Sledeći (*next*) i prethodni (*prev*)

- Premešta iterator na poziciju sledećeg `next()`, odnosno prethodnog elementa `prev()`
- Premeštanje može biti za jedan ili za zadani broj elemenata (pozicija)
- Za bidirekcione iteratore, zadani broj pozicija može biti *negativan*, za premeštanje u suprotnom smeru



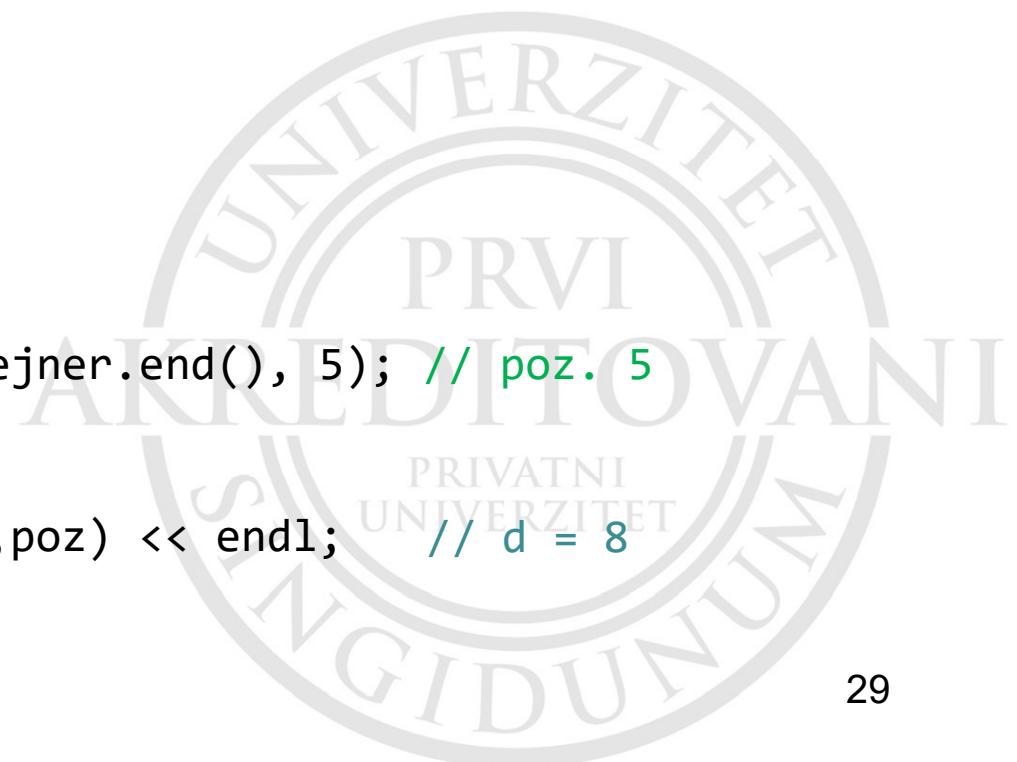
4.3 Razmak (*distance*)

- Funkcija `distance()` vraća razliku pozicija dva iteratora u okviru jednog kontejnera (razmak, rastojanje)
- Npr.

```
list<int> kontejner;
// Umetanje elemenata -3 do 9
for (int i=-3; i<=9; ++i) {
    kontejner.push_back(i);
}

// Pozicija elementa 5
list<int>::iterator poz;
poz = find(kontejner.begin(), kontejner.end(), 5); // poz. 5

// Razmak -3..5
cout << distance(kontejner.begin(), poz) << endl; // d = 8
```



4.4 Zamena (*iter_swap*)

- Funkcija omogućava međusobnu zamenu vrednosti elemenata jednog kontejnera na pozicijama na koje pokazuju dva iteratora
- Npr. zamena vrednosti prvog i drugog elementa
`iter_swap(kontejner.begin(), next(kontejner.begin()));`



5. Iteratorski adapteri

1. Iteratori umetanja (*insert*)
2. Iteratori tokova (*stream*)
3. Reverzni iteratori (*reverse*)
4. Iteratori premeštanja (*move*)



5.1 Iteratori umetanja (*insert*)

- Iteratori umetanja (*inserters*) zasnivaju se na standardnim iteratorima, koji pristupaju *postojećim* elementima kontejnera, ali se koriste za dodavanje *novih* elemenata
- Inserteri umesto prepisivanja elementa *dodaju* novi element
 - `back_insert_iterator` dodaje nove elemente na kraj kontejnera koji imaju metod `push_back()`
 - `front_insert_iterator` dodaje nove elemente na početak kontejnera koji imaju metod `push_front()`
 - `insert_iterator` se može koristiti za umetanje elemenata na bilo koje mesto u kontejnere koji imaju funkciju člana `insert()`

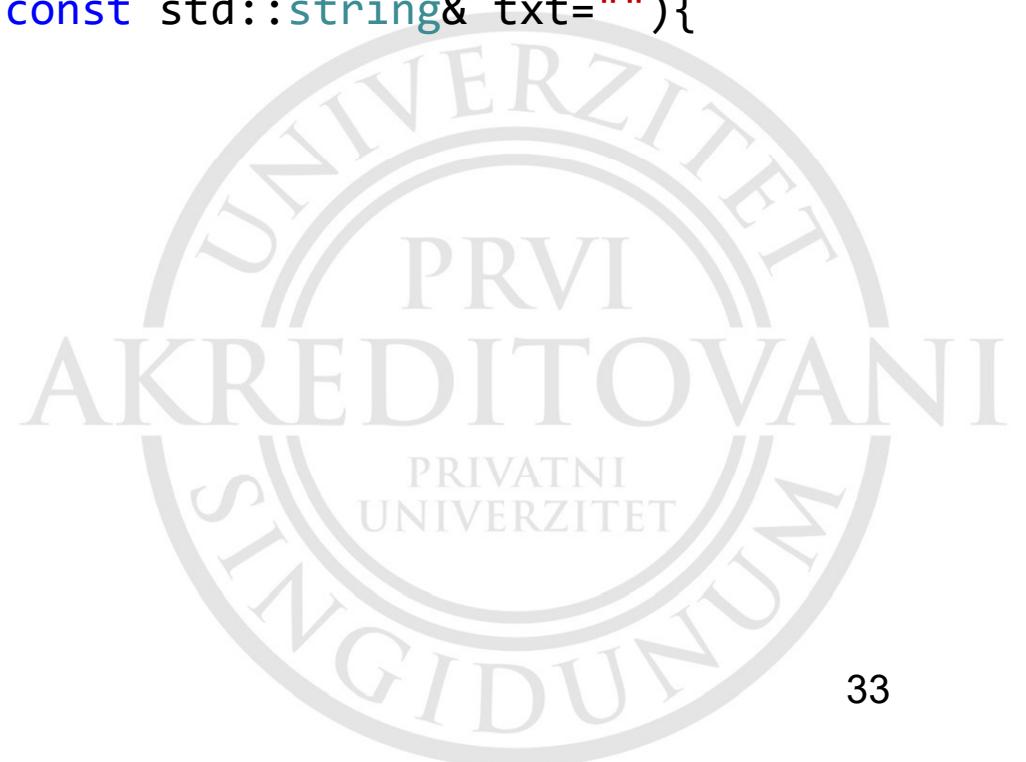
Expression	Effect
<code>*iter</code>	No-op (returns <code>iter</code>)
<code>iter = value</code>	Inserts <code>value</code>
<code>++iter</code>	No-op (returns <code>iter</code>)
<code>iter++</code>	No-op (returns <code>iter</code>)

Primer: Opšti iterator umetanja (inserter)

```
#include <set>
#include <list>
#include <algorithm>
#include <iterator>
using namespace std;

template <typename T>
inline void printElems(const T& kont, const std::string& txt=""){
    std::cout << txt;
    for (const auto& elem : kont) {
        std::cout << elem << ' ';
    }
    std::cout << std::endl;
}

int main(){
    set<int> kont;
```

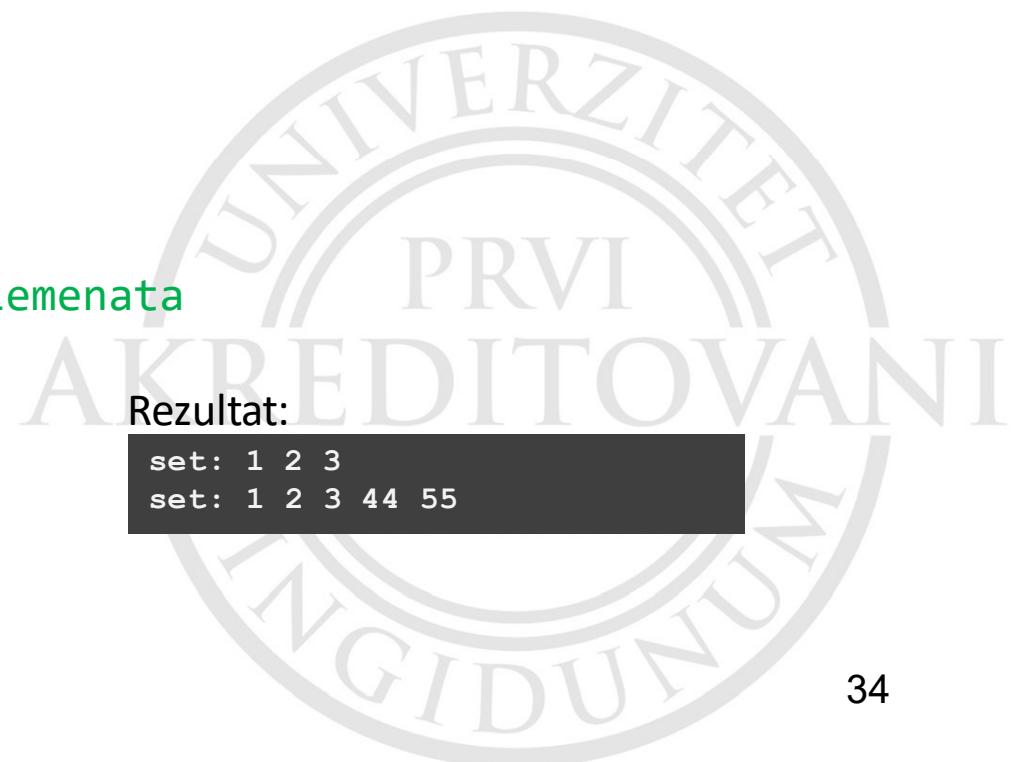


Primer: Opšti iterator umetanja (inserter)

```
// Kreiranje insert iteratora za kontejner kont
// (neodgovarajući način)
insert_iterator<set<int>> iter(kont, kont.begin());
```

```
// Umetanje elemenata s uobičajenim intefejsom iteratora
*iter = 1;
iter++;
*iter = 2;
iter++;
*iter = 3;
printElems(kont, "set: ");
```

```
// Kreiranje insertera i umetanje elemenata
// (odgovarajući način)
inserter(kont, kont.end()) = 44;
inserter(kont, kont.end()) = 55;
printElems(kont, "set: ");
```



Primer: Opšti iterator umetanja (inserter)

```
// Upotreba insertera za umetanje svih elemenata u listu
list<int> kont2;
copy (kont.begin(), kont.end(),           // koji elementi
      inserter(kont2, kont2.begin())); // gde se kopiraju
printElms(kont2, "list: ");

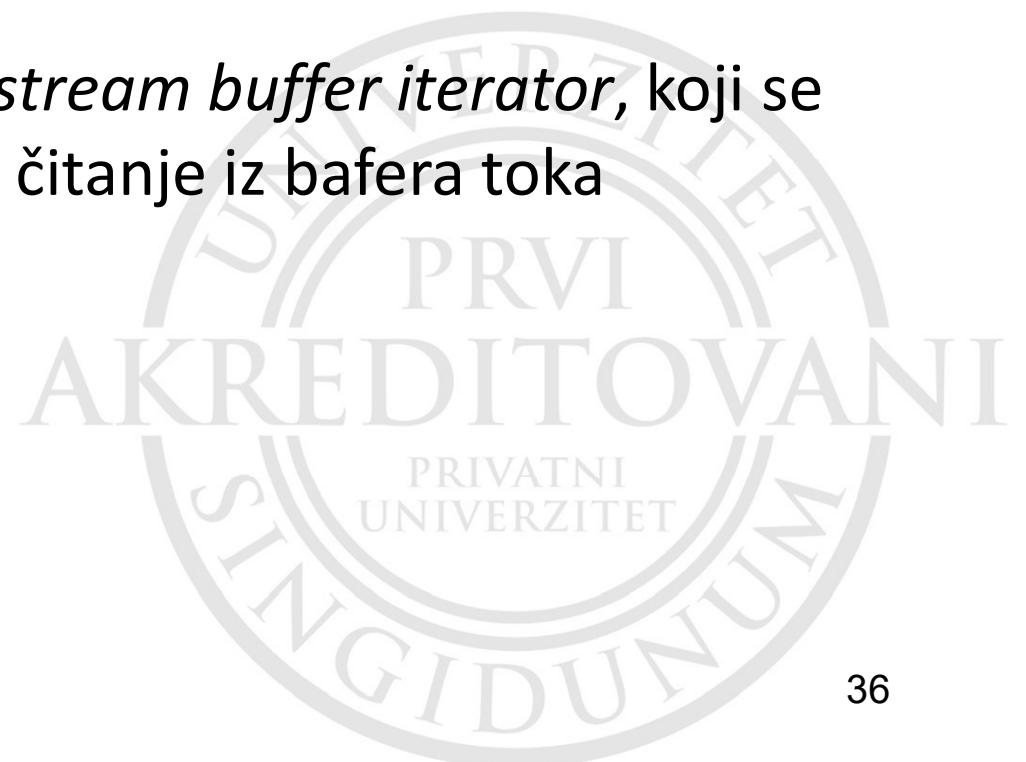
// Upotreba insertera za ponovno umetanje svih elemenata
// u listu ispred drugog elementa
copy (kont.begin(), kont.end(),           // koji elementi
      inserter(kont2, ++kont2.begin())); // gde se kopiraju
printElms(kont2, "list: ");
}
```

Rezultat:

```
set: 1 2 3
set: 1 2 3 44 55
list: 1 2 3 44 55
list: 1 1 2 3 44 55 2 3 44 55
```

5.2 Iteratori tokova (*stream*)

- Iteratori tokova su iteratorski adapteri koji omogućavaju korišćenje tokova kao izvor ili odredište algoritama
- Iterator ulaznog toka `istream` može se koristiti za čitanje elemenata iz ulaznog toka, a iterator izlaznog toka `ostream` za upis vrednosti u izlazni tok
- Posebna vrsta iteratora toka je *stream buffer iterator*, koji se može koristiti za direktni upis ili čitanje iz bafera toka



5.3 Reverzni iteratori (*reverse*)

- Reverzni iteratori redefinišu operatore inkrementa i dekrementa da promene smer u odnosu na obične iteratore, tako da algoritmi obrađuju elemente obrnutim redosledom
- Sve kontejnerske klase osim jednosmernih lista i neuređenih kontejnera omogućavaju upotrebu reverznih iteratora, npr.

```
void print (int elem) { cout << elem << ' ' }

int main() {

    list<int> kont = {1,2,3,4,5,6,7,8,9}; // lista elemenata
    // Prikaz svih elemenata u normalnom poretku pomoću alg.
    // for_each(range, fun) i korisničke funkcije print(elem)
    for_each(kont.begin(), kont.end(), print);
    cout << endl;
    // Prikaz u obrnutom poretku zamenom iteratora reverznim
    for_each(kont.rbegin(), kont.rend(), print);
    cout << endl;
}
```

5.4 Iteratori premeštanja (*move*)

- Iteratori premeštanja su iteratorski adapteri koji svaki pristup izabranom elementu pretvaraju u operaciju *promeštanja*, npr.
 - kopiranje stringa s u string v1

```
std::list<std::string> s;
std::vector<string> v1(s.begin(), s.end());
```
 - premeštanje (*move*) stringa s u string v2

```
std::vector<string> v2(make_move_iterator(s.begin()),
                         make_move_iterator(s.end()));
```
- Jedna od primena ovih iteratora je da omoguće algoritmima da umesto kopiranja *promeštaju* elemente iz zadanog opsega
- Pri tome treba obezrediti da se svakom elementu pristupa samo jednom

6. Upotreba iteratora

- Prenosivost koda s iteratorima
- Korisnički definisani iteratori



Prenosivost koda s iteratorima

- Prenosivost koda s iteratorima ostvaruje se izbegavanjem eksplisitne upotrebe operatora inkrementa i dekrementa
- Npr. za strukturu

```
std::vector<int> kontejner;
```

radi sortiranja od *drugog* elementa, umesto operatora `++`

```
// Sortiranje od drugog elementa
if (kontejner.size() > 1) {
    std::sort(++kontejner.begin(), kontejner.end());
}
```

bolje je koristiti funkciju `next()`

```
// Sortiranje od drugog elementa
if (kontejner.size() > 1) {
    std::sort(std::next(kontejner.begin()), kontejner.end());
}
```

Korisnički definisani iteratori

- Korisnički definisani iteratori kreiraju se kao objekti posebne strukture i svojstava, koji nasleđuju osobine odgovarajućih postojećih iteratora, npr.

```
namespace std {  
    template <typename T>  
        struct iterator_traits {  
            typedef typename T::iterator_category iterator_category;  
            typedef typename T::value_type value_type;  
            typedef typename T::difference_type difference_type;  
            typedef typename T::pointer pointer;  
            typedef typename T::reference reference;  
        };  
}
```

- Generičke funkcije omogućavaju prilagođavanje tipova elemenata različitih kontejnera

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Josuttis N. M. , *The C++ Standard Library*, 2nd Ed, Pearson Education, 2012
8. Web izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Vikipedija www.wikipedia.org
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 11

Tokovi podataka, ulaz-izlaz i rad s fajlovima u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Pojam toka podataka
3. Rad s fajlovima
4. Formatiranje ulaza i izlaza
5. Formatiranje pomoću funkcija klase ios
6. Upotreba ulazno-izlaznih manipulatora
7. Modul <format>
8. Primeri programa



1. Uvod

- Ulaz-izlaz podataka u jeziku C++
- Formatiranje podataka u ulazno-izlaznim operacijama



Ulaz-izlaz podataka u jeziku C++

- Ulaz i izlaz podataka može biti npr.
 - čitanje i upis podataka u tekstualne datoteke ili bazu podataka
 - prikaz podataka u grafičkom prozoru aplikacije
 - komunikacija putem mreže
- Apstrakcija ulazno-izlaznih uređaja u jeziku C++ omogućava *jedinstveni interfejs* ovih operacija
- Jezik C++ nema ugrađene naredbe za ulaz i izlaz podataka, već se oslanja na ulazno-izlazne operacije iz biblioteke klasa, koje su opisane u zaglavlju **<iostream>** i dostupne kroz imenik std
 - stariji format biblioteke, opisan u zaglavlju **iostream.h**, bio je dostupan u globalnom imeniku

Formatiranje podataka u ulazno-izlaznim operacijama

- Biblioteka klasa `<iostream>` omogućava formatirani upis i čitanje (tekstualnih) podataka
- Format podataka u ulazno-izlaznim operacijama može se precizno definisati na više načina
 1. Pomoću *flegova (flags)* i drugih vrednosti pomoću kojih se upravlja formatiranjem podataka određenog toka
 2. Pomoću posebnih *manipulatorskih funkcija* prilikom operacija umetanja i izdvajanja (`>>` i `<<`)
 3. Pomoću *regularnih izraza* prilikom čitanja (parsiranja) podataka
 4. Pomoću modula `<format>` koji je uveden u verziji C++20

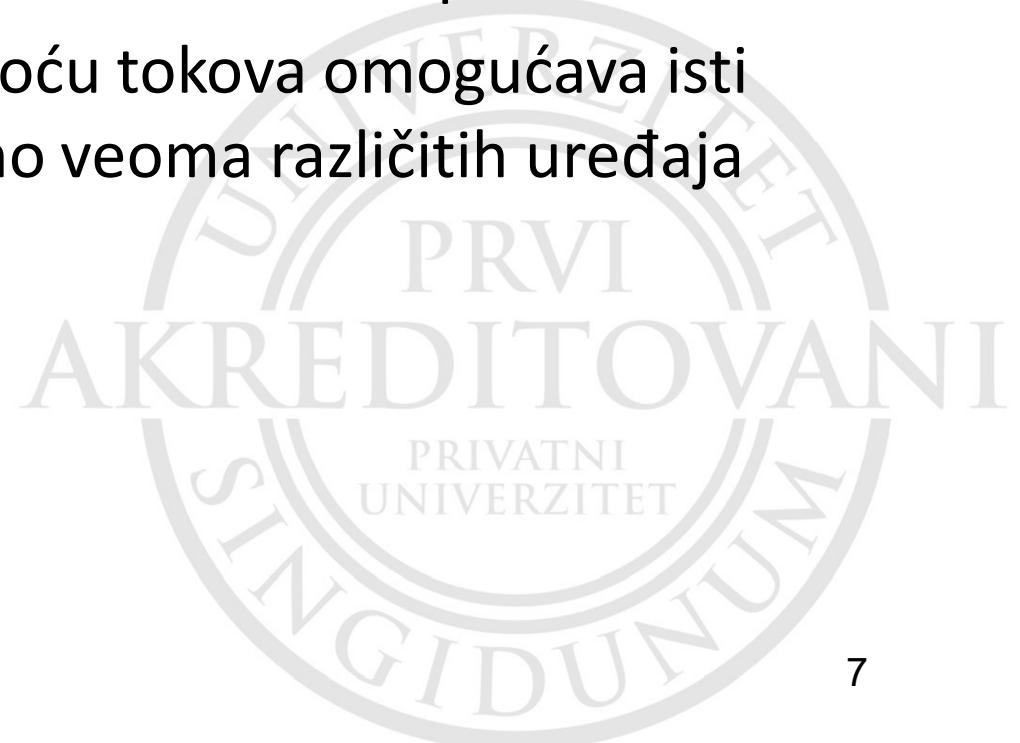
2. Pojam toka podataka

- 1. Pojam toka**
- 2. Vrste tokova**
- 3. Predefinisani tokovi**
- 4. Klase tokova**



2.1 Pojam toka

- **Tok** podataka (*stream*) je pogodna apstrakcija ulaznih ili izlaznih uređaja sistema, koja omogućava isti način rada s različitim fizičkim uređajima
- Tok je *logički interfejs* za datoteke, koje fizički mogu biti npr. disk, tastatura, ekran računara, komunikacioni port i sl.
- Jedinstveni prikaz uređaja pomoću tokova omogućava isti način programiranja međusobno veoma različitih uređaja



2.2 Vrste tokova

- Osnovne kategorije tokova definisane su u odnosu na tip podataka, koji mogu biti *binarni* i *tekstualni*
- **Binarni** podaci se zapisuju unutar računara u binarnom obliku bez ikakve transformacije i ljudima nisu direktno čitljivi
- **Tekstualni** podaci su nizovi znakova, ljudima direktno čitljivi, koji se u memoriji računara zapisuju u nekom binarnom formatu, npr. Unicode ili ASCII kodu
 - binarni tok **10100111** može da predstavlja decimalni broj **167** tipa int
 - tekstualni tok za zapis broja 167 je niz znakova "**1**", "**6**" i "**7**", koji se binarno kodiraju kao **0x31**, **0x36** i **0x37**, odnosno binarno **00110001**, **00110110** i **00110111**
- Binarni podaci su direktno *prenosivi* između različitih sistema

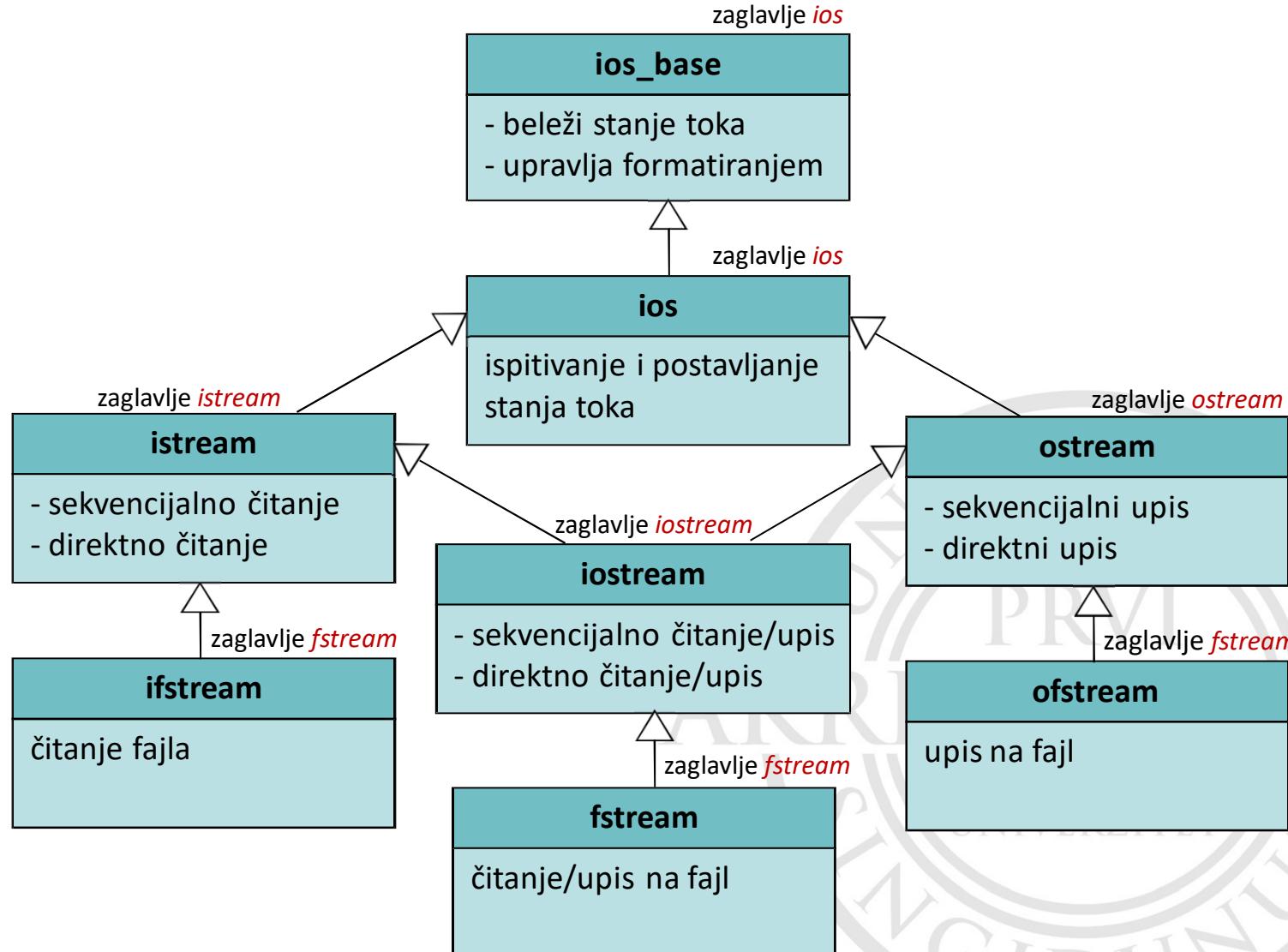
2.3 Predefinisani tokovi

- Biblioteka klasa sadrži predefinisane objekte tokova *cin*, *cout*, *cerr* i *clog*, koji su dostupni na početku izvršavanja programa
- Objekt toka **cin** je pridružen standardnom ulazu, uređaju tipa tastature
- Objekt toka **cout** je pridružen standardnom izlazu, ekranu računara
- Objekti toka **cerr** i **clog** takođe su pridruženi standardnom izlazu, ekranu računara i služe za ispis poruka o greškama
 - objekt *cerr* nije baferovan i odmah prikazuje poruke, dok je *clog* baferovan i prikaz informacija se vrši tek po punjenju bafera
- Ovi tokovi se mogu *preusmeravati* na različite uređaje ili datoteke

2.4 Klase tokova

- Ulazno-izlazne operacije se oslanjaju na dve različite hijerarhije klasa tokova:
 - `basic_streambuf` - ulazno izlazne operacije niskog nivoa, klasa koja predstavlja osnovu sistema i retko se koristi direktno
 - `basic_ios` - ulazno izlazne operacije visokog nivoa: formatiranje, provera grešaka i stanja U/I operacija
- Klase izvedene iz klase `basic_ios` su klase ulaznih, izlaznih i ulazno-izlaznih tokova `basic_istream`, `basic_ostream` i `basic_iostream`
- Odgovarajuće klase, koje se najčešće koriste u programima su `streambuf`, `ios`, `istream`, `ostream`, `iostream` i klase za datoteke `ifstream`, `ofstream` i `fstream`

Ilustracija: Hijerarhija klasa Ul tokova visokog nivoa



3. Rad s fajlovima

1. Pristup fajlovima
2. Čitanje tekstualnog fajla
3. Upis na tekstualni fajl
4. Objekti i tokovi podataka
5. Preklapanje ulaznih i izlaznih operatora



3.1 Pristup fajlovima

- Pristup fajlovima u jeziku C++ ostvaruje se pomoću klasa opisanih u zaglavlju **<fstream>**
- Pristup fajlu ostvaruje se povezivanjem s tokom odgovarajućeg tipa: ulaznim, izlaznim ili ulazno-izlaznim, npr.

```
ifstream in;    // ulazni tok
ofstream out;   // izlazni tok
fstream io;     // ulazno-izlazni tok
```

- Povezivanje se vrši odgovarajućom funkcijom *open()*:

```
void ifstream::open(const char *filename, ios::openmode
mode=ios::in);
void ofstream::open(const char *filename, ios::openmode
mode=ios::out);
void fstream::open(const char *filename, ios::openmode
mode=ios::in|ios::out);
```

Otvaranje fajla

- Promenljiva *filename* može biti relativna ili absolutna putanja fajla, npr.

```
ofstream outfile;  
outfile.open("podaci.dat"); // relativna putanja  
outfile.open("C:\\\\folder\\\\podaci.dat"); // absolutna putanja
```

- Parametar *mode* određuje način otvaranja fajla:

Parametar mode	Značenje
ios::app	Sadržaj se dodaje na kraj postojeće datoteke.
ios::ate	Ako datoteka već postoji, program se pomera direktno na njen kraj. Može se upisivati bilo gde u datoteku (ovaj režim se obično koristi sa binarnim datotekama)
ios::binary	Sadržaj se upisuje u datoteku u binarnom obliku, a ne u tekstualnom (koji je podrazumevani)
ios::in	Sadržaj se čita iz datoteke. Ako datoteka ne postoji, neće biti napravljena.
ios::out	Sadržaj se upisuje u datoteku, a ako ona već ima sadržaj, prepisuje se.
ios::trunc	Ako datoteka već postoji, njen sadržaj će biti prepisan (podrazumevani režim za ios::out)

Otvaranje i zatvaranje fajla

- Funkcija `open` vraća vrednost koja u logičkim izrazima ima vrednost `false` ako otvaranje ne uspe, npr.

```
if (!tok) {  
    cout << "Greška: fajl se ne može otvoriti!" << endl;  
};
```

- Uspešnost otvaranja može se proveriti i funkcijom `is_open()`

```
if (tok.isopen()) {  
    cout << "Fajl je otvoren!" << endl;  
}
```

- Fajl se zatvara pomoću `tok.close()`. Funkcija `open()` nije obavezna, jer klase `fstream`, `istream` i `ostream` imaju konstruktore, koji automatski otvaraju fajlove, npr.

```
ifstream tok("mojFajl"); // automatsko otvaranje
```

3.2 Čitanje tekstualnog fajla

- Upis i čitanje fajlova vrši se pomoću standardnih operatora *umetanja* i *izdvajanja* za tokove povezane s fajlom
- Fajl koji se otvara za čitanje mora prethodno da postoji
- Provera kraja datoteke vrši se pomoću funkcije `eof()`, člana klase `istream`
- Čitanje fajlova vrši se pomoću operatora *izdvajanja* (`>>`) ili funkcije `getline()`
 - operator izdvajanja čita podatke do prvog delimitera (beline, *whitespace*) kao što su razmak, tabulator i sl.
 - funkcija `getline()` čita podatke iz fajla *red po red*

Funkcija getline

- Funkcija `getline()` je član svih klasa ulaznih tokova
- Prototipovi ove funkcije su

```
istream &getline(char *buf, streamsize num)
```

```
istream &getline(char *buf, streamsize num, char delim)
```

- Prva varijanta funkcije učitava podatke dužine do *num*-1 znakova ili dok ne najde na znak kraja reda ili kraj fajla
- Druga verzija kao delimiter koristi aktuelni argument *delim*
- Znak kraja reda i delimiter se ne učitavaju, već se na kraj učitanog niza znakova upisuje *nula*

3.3 Upis u tekstualni fajl

- Upis na *tekstualni* fajl vrši se pomoću standardnih operatora umetanja (<<)
- Upis na fajl je *baferovan*, tako da se podaci fizički upisuju tek kad se interni bafer napuni
- Pošto se prilikom otkaza sistema podaci iz radne memorije mogu izgubiti, upis na fajl može se aktivirati ranije funkcijom **flush()** čiji je prototip

```
ostream &flush();
```



Primer: Upis i čitanje tekstualnog fajla(1/2)

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

int main () {
    char bafer [200];
    cout << "Unesite ime i adresu: ";
    cin.getline(bafer, 200);                                // čitanje standardnog ulaza
    cout << endl;

    ofstream outfile("ime.txt"); // konstruktor klase ostream
    if (!outfile) {
        cerr << endl;
        cerr << "Greska: fajl se ne moze otvoriti "
            << "u rezimu dodavanja!" << endl;
        return 1;
    }
```



Primer: Upis i čitanje tekstualnog fajla (2/2)

```
outfile << bafer << endl;           // ispis/dodavanje teksta u outfile
outfile.close();                      // zatvaranje oufile

ifstream infile("ime.txt");           // konstruktor klase istream
while (!infile.eof()) {
    infile.getline(bafer, 200);        // čitanje teksta s fajla istream
    cout << bafer << endl;            // ispis teksta na standardni izlaz
}
infile.close();                        // zatvaranje fajla istream

return 0;
}
```

Unesite ime i adresu: Petar Petrović Poenkareova 7a

Petar Petrović Poenkareova 7a

Unesite ime i adresu: Ivana Ivanović Trešnjina 14

Petar Petrović Poenkareova 7a
Ivana Ivanović Trešnjina 14

3.4 Objekti i tokovi

- Proces upisa objekta neke klase u tok, tako da može biti ponovo vraćen u memoriju, naziva se *serijalizacija*
- Vraćanje objekta iz toka u memoriju naziva se *deserijalizacija*
- Članovi podaci jedne klase mogu biti osnovni tipovi, klase i pokazivači, tako da su ulazne i izlazne operacije za objekte *specifične* za svaku klasu
- Definišu se nove (nadjačane) verzije operatora *umetanja* i *izdvajanja* za određenu klasu kao prijateljske funkcije klase

3.5 Preklapanje ulaznih i izlaznih operatora

- Operatori `<<` i `>>` mogu se preklapati na isti način kao i binarni aritmetički operatori
- Standardna biblioteka obezbeđuje preklopljene definicije ulaznih i izlaznih tokova za standardne tipove vrednosti:

```
ostream& operator<<(ostream& out, tip vrednost);
```

- Upotreba preklopljenih operatora omogućava proširenje ulazno-izlaznih operacija na *nove tipove* podataka, npr.

```
ostream& operator<<(ostream& out, const Razlomak& v) {  
    out << v.brojilac() << "/" << v.imenilac();  
    return out;  
}
```

4. Formatiranje ulaza i izlaza

- 1.** Format podataka u ulazno-izlaznim operacijama
- 2.** Upotreba regularnih izraza



4.1 Format podataka u ulazno-izlaznim operacijama

- Format podataka u ulazno-izlaznim operacijama može se precizno definisati
 1. Pomoću metoda klase `ios`, koji se koriste za postavljenje flegova i drugih vrednosti kojima se upravlja formatiranjem podataka toka
 2. Pomoću posebnih manipulatorskih funkcija, koje se koriste uz operatore umetanja i izdvajanja (`>>` i `<<`)
 3. Pomoću modula `<format>` od verzije C++20
- Navedeni metodi omogućavaju precizno definisanje formata izlaznih podataka, dok se rad s podacima prilikom njihovog čitanja (parsiranje) može olakšati upotrebom regularnih izraza
 - u mnogim primenama *performanse* upotrebe regularnih izraza znatno zaostaju za standardnim metodima formatiranja

4.2 Upotreba regularnih izraza

- U jeziku C++ rad sa stringovima može se pojednostaviti upotrebom *regularnih izraza*, koji omogućavaju skraćeni opis nizova znakova, koji su formirani u skladu s nekim formalnim pravilima (jezikom)
 - standardni zapis regularnih izraza razvijen je 1950-tih i kompletiran 1970-tih za potrebe razvoja Unix programa
 - standardna biblioteka jezika C++ podržava više varijanti sintakse regularnih izraza; podrazumevajuća varijanta je istovremeno i deo standarda jezika *JavaScript* (ECMAScript)
- Regularni izrazi su uključeni u STL biblioteku od verzije C++11
- Upotreba regularnih izraza u jeziku C++ omogućena je preko zaglavlja <**regex**>

Primer: Upotreba regularnih izraza u jeziku C++

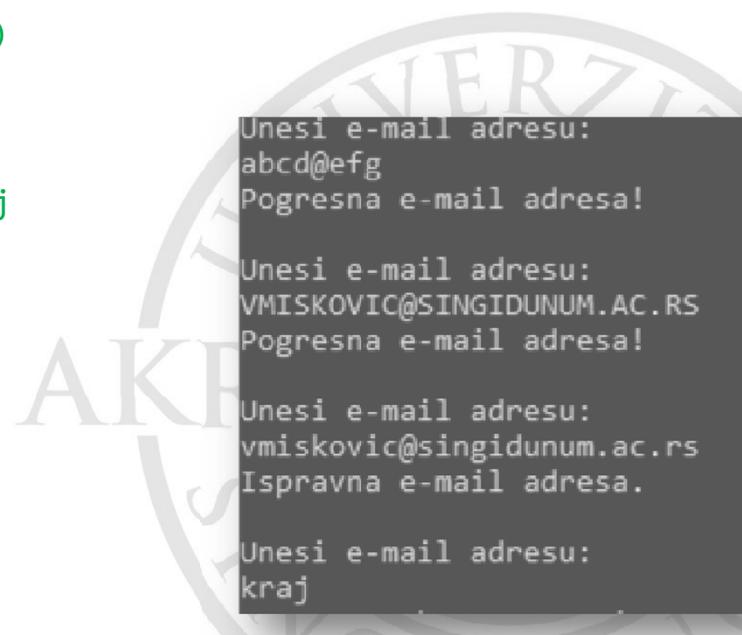
```
#include <iostream>
#include <regex>
#include <string>
using namespace std;

int main() {

    string eadresa;
    regex email("^[a-z0-9._%+-]+@[a-z0-9.-]+\\.[a-z]{2,6}$");

    // Provera ispravnosti e-mail adrese (mala slova)
    while(true) {
        cout << "Unesi e-mail adresu: " << endl;
        cin >> eadresa;
        if (eadresa=="kraj") // ako korisnik unese kraj
            break;

        if (regex_match(eadresa,email))
            cout << "Ispravna e-mail adresa.\\" << endl;
        else {
            cout<< "Pogresna e-mail adresa!\\" <<endl;
        }
    }
    return 0;
}
```



```
Unesi e-mail adresu:
abcd@efg
Pogresna e-mail adresa!

Unesi e-mail adresu:
VMISKOVIC@SINGIDUNUM.AC.RS
Pogresna e-mail adresa!

Unesi e-mail adresu:
vmiskovic@singidunum.ac.rs
Ispravna e-mail adresa.

Unesi e-mail adresu:
kraj
```

5. Formatiranje pomoću funkcija klase ios

1. Formatiranje pomoću funkcija člana klase ios
2. Upotreba flegova za formatiranje
3. Preciznije podešavanje formata



5.1 Formatiranje pomoću funkcije člana klase ios

- Formatiranje metodima klase *ios* koristi skup flegova `fmtflags`, kojim se upravlja formatiranjem podataka toka, npr.
 - `left` - označava levo poravnavanje izlaza (`right` je desno)
 - `oct, hex, dec` - prikaz vrednosti u različitim brojnim sistemima
 - `showbase` - flag za prikaz oznake brojnog sistema
 - `uppercase` - prikaz se vrši velikim slovima
 - `boolalpha` - logičke vrednosti se mogu unositi i ispisivati simbolički kao *true* i *false*
- Kada nije postavljen nijedan fleg, format bira prevodilac

5.2 Upotreba flegova za formatiranje

- Postavljanje flegova vrši se metodom `setf()` klase `ios`, npr.

```
tok.setf(ios::showpos); // ispis znaka + za poz. vred.
```

- Nazivi flegova su definisani kao konstante u klasi `ios`, pa se za pristup koristi operator dosega
- U jednoj naredbi moguće je istovremeno definisati više flegova, npr.

```
cout.setf(ios::scientific | ios::showpos);
```

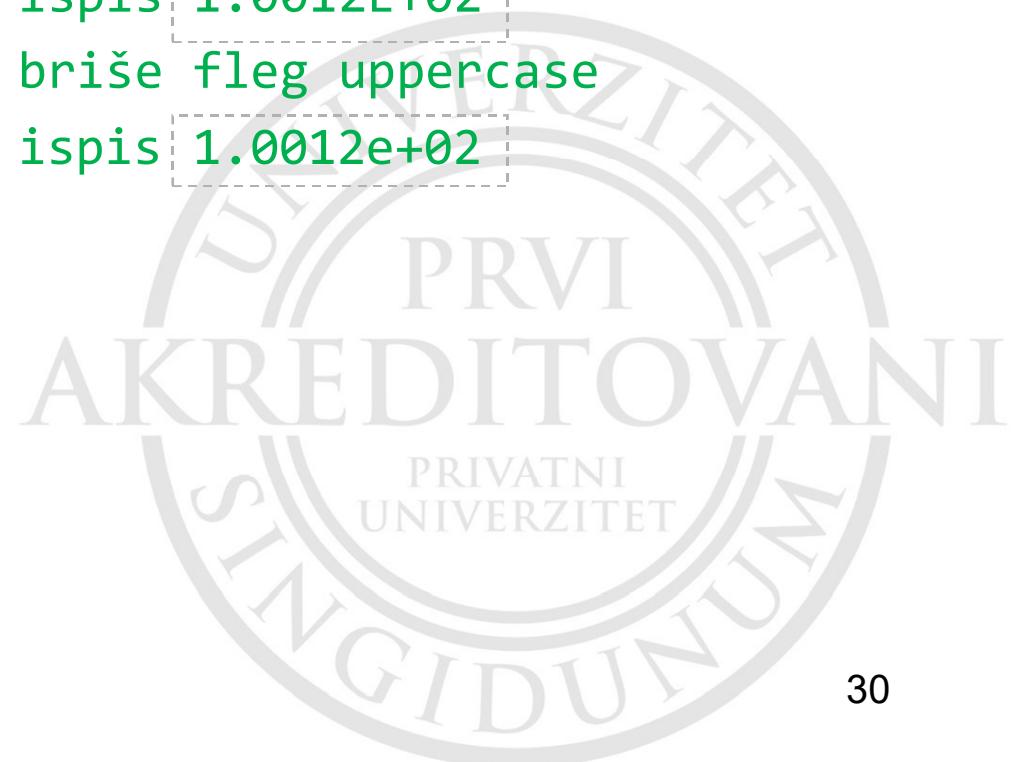
- Brisanje flega vrši se metodom `unsetf()`, a trenutne vrednosti pojedinih flegova dobijaju se pomoću funkcije `flags()` definisane kao

```
void unsetf(fmtflags flags);
fmtflags flags();
```

Primer: Formatiranje izlaza pomoću flegova

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

int main () {
    cout.setf(ios::uppercase | ios::scientific);
    cout << 100.12;                      // ispisi 1.0012E+02
    cout.unsetf(ios::uppercase);          // briše fleg uppercase
    cout << endl << 100.12;              // ispisi 1.0012e+02
    return 0;
}
```



Preciznije podešavanje formata

- Klasa `ios` poseduje funkcije `width()` za preciznije podešavanje širine polja ispisa, `precision()` za preciznost ispisa brojeva i `fill()` za izbor znak za popunu, čiji su prototipovi:

```
streamsize width(streamsize w); // minimalna širina
streamsize precision(streamsize p); // p inicialno 6
char fill(char ch); // ch je novi, a vraća stari znak
```
- Funkciju `width()` je potrebno pozvati pre svake izlazne operacije
- Ako je ispis kraći, popunjava se znakom za popunu, koji je inicialno razmak
- Funkcijom `precision()` definiše se broj cifara u prikazu brojeva u pokretnom zarezu

Primer: Preciznije podešavanje formata

```
#include <iostream>
using namespace std;
int main () {
    cout.precision(4);
    cout.width(10);
    cout << 10.12345 << endl; // prikaz 10.12
    cout.fill('*');
    cout.width(10);
    cout << 10.12345 << endl; // prikaz *****10.12
    cout.width(10);
    cout << "Zdravo!" << endl; // prikaz ***Zdravo!
    cout.width(10);
    cout.setf(ios::left);      // levo pravnanje ispisa
    cout << 10.12345 << endl; // prikaz 10.12 ****
    return 0;
}
```

6. Upotreba ulazno-izlaznih manipulatora

- 1.** Ulazno izlazni manipulatori
- 2.** Manipulatorske funkcije
- 3.** Argumenti manipulatorskih funkcija
- 4.** Definicija manipulatorske funkcije



6.1 Ulazno izlazni manipulatori

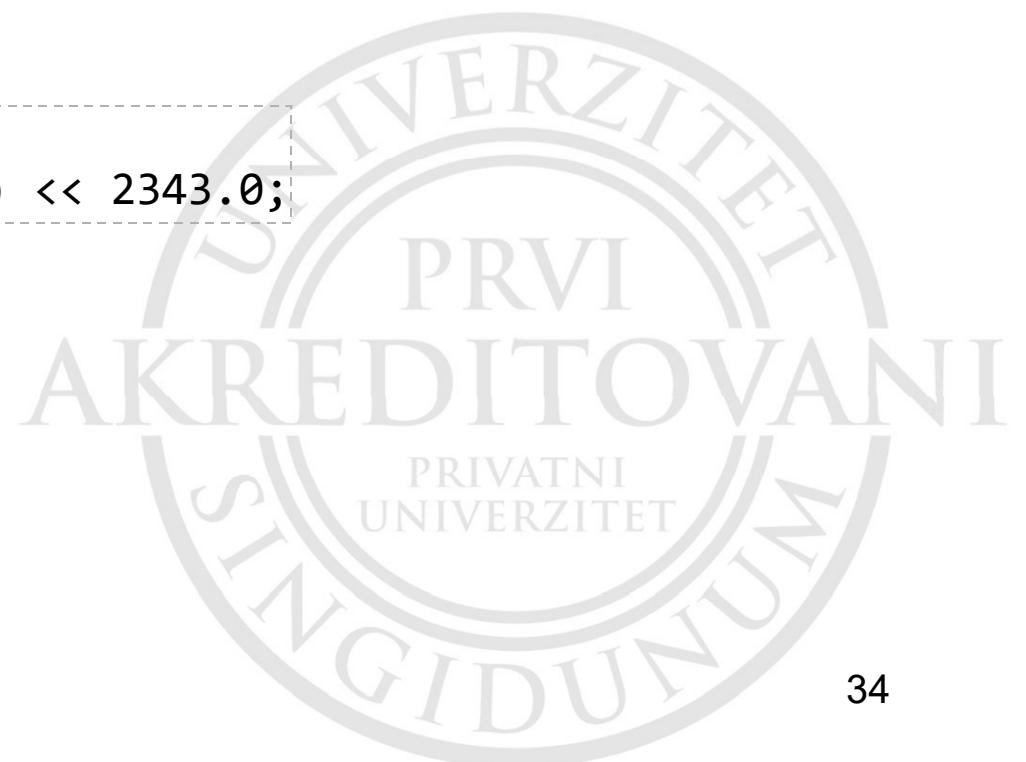
- Ulazno-izlazni format se može definisati pomoću posebnih *manipulatorskih* funkcija definisanih u zaglavljtu <iomanip>
- Npr. rezultat izvršavanja ulazno-izlaznih naredbi programa

```
#include <iostream>
#include <iomanip>
using namespace std;
int main () {
    cout << hex << 100 << endl;
    cout << setfill('?)' ) << setw(10) << 2343.0;
    return 0;
}
```

je ispis

64

??????2343



6.2 Manipulatorske funkcije

- Manipulatorske funkcije su posebne funkcije namenjene upotrebi uz operator umetanja << i izdvajanja >>
 - menjaju prametre formatiranja tokova i umeću ili izdvajaju određene specijalne znakove
- Mogu se koristiti i kao standardne funkcije, čiji je argument objekt tipa toka, npr.
`boolalpha(cout);`



Manipulator	Namena	U/I
boolalpha	Postavlja fleg boolalpha	ulazno/izlazni
dec	Postavlja fleg dec	ulazno/izlazni
endl	Prelazi u novi red i prazni tok	izlazni
ends	Prikazuje null	izlazni
fixed	Uključuje fleg fixed	izlazni
flush	Prazni tok	izlazni
hex	Postavlja fleg za heksadecimalni prikaz	izlazni
internal	Postavlja fleg internal	izlazni
left	Postavlja fleg left	izlazni
noboolalpha	Isključuje fleg noboolalpha	ulazno/izlazni
noshowbase	Isključuje fleg noshowbase	izlazni
noshowpoint	Isključuje fleg noshowpoint	izlazni
noshowpos	Isključuje fleg noshowpos	izlazni
noskipws	Isključuje fleg noskipws	ulazni
nounitbuf	Isključuje fleg nounitbuf	izlazni
nocase	Isključuje fleg nouppercase	izlazni
oct	Postavlja fleg za oktalni prikaz	ulazno/izlazni
resetiosflags (fmtflags f)	Resetuje flegove navedene u f	ulazno/izlazni
right	Postavlja fleg right	izlazni
scientific	Uključuje eksponencijalni prikaz	izlazni
set base(int base)	Postavlja brojnu osnovu na base	ulazno/izlazni
setfill(int ch)	Postavlja znak za popunu na ch	izlazni
setiosflags(fmtflags f)	Postavlja flegove navedene u f	ulazno/izlazni
setprecision(int p)	Postavlja broj znakova za preciznost	izlazni
setw(int w)	Postavlja širinu polja na w	izlazni
showbase	Postavlja fleg showbase	izlazni
showpoint	Postavlja fleg showpoint	izlazni
showpos	Postavlja fleg showpos	izlazni
skipws	Postavlja fleg skipws	ulazni
unitbuf	Postavlja fleg unitbuf	izlazni
uppercase	Uključuje fleg uppercase	izlazni
ws	Preskače vodeće razmake	ulazni

Manipulatorske funkcije (pregled)

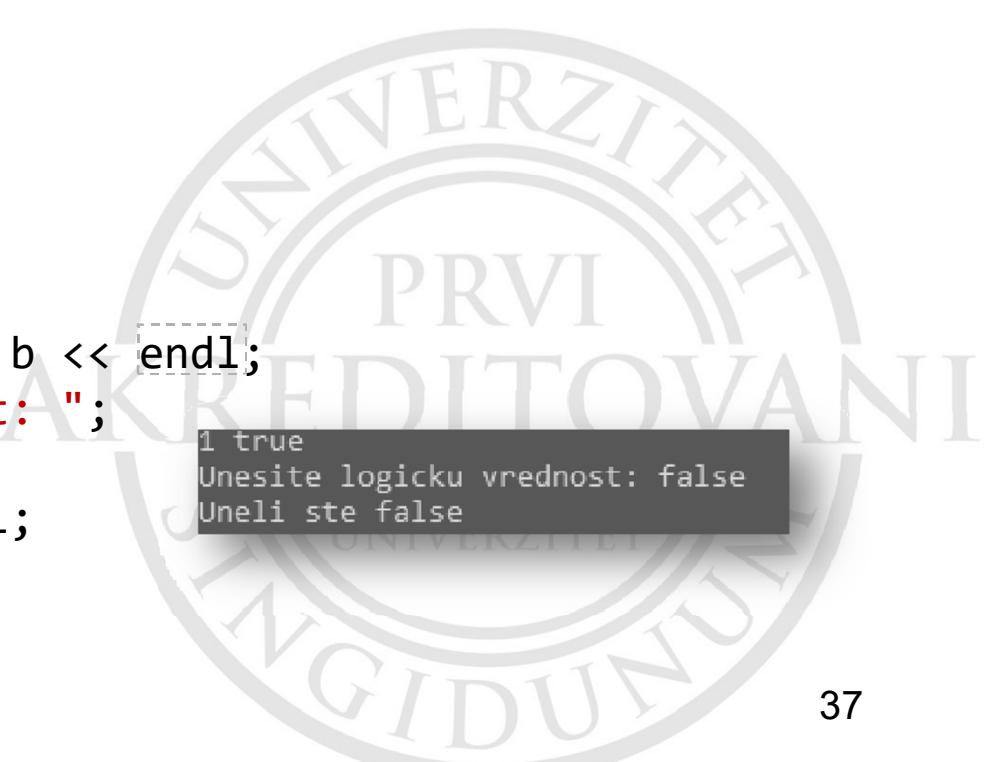


6.3 Argumenti manipulatorskih funkcija

- Upotrebom manipulatorskih funkcija dobija se kompaktniji i pregledniji kod ulazno-izlaznih izraza
- Funkcije mogu biti s argumentima ili bez argumenata, kada se u izrazima izostavljaju zagrade, npr. za boolalpha() i endl()

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

int main () {
    bool b;
    b = true;
    cout << b << " " << boolalpha << b << endl;
    cout << "Unesite logicku vrednost: ";
    cin >> boolalpha >> b;
    cout << "Uneli ste " << b << endl;
    return 0;
}
```



```
1 true
Unesite logicku vrednost: false
Uneli ste false
```

6.4 Definicija manipulatorske funkcije

- Manipulatorske funkcije su funkcije čiji je argument *referenca na tok* koje se koriste za promenu parametara formatiranja ili umetanje ili izdvajanje specijalnih znakova
- Opšti oblik manipulatorske funkcije je

```
ostream &naziv_funkcije(ostream &stream) {  
    // telo manipulatorske funkcije  
    return stream;  
}
```

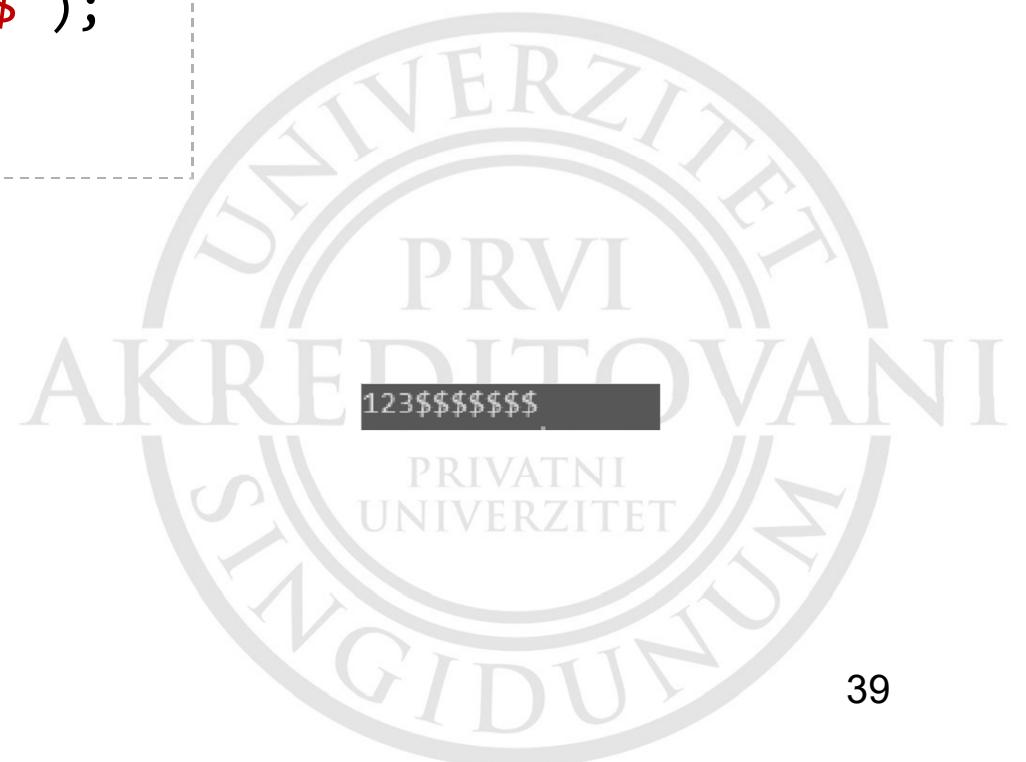
- Argument se u pozivu manipulatorske funkcije ne navodi kad se ona poziva za tok

Primer: Definisanje sopstvene manipulatorske funkcije

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

ostream &setup(ostream &stream) {
    stream.setf(ios::left);
    stream << setw(10) << setfill('$');
    return stream;
}

int main () {
    cout << setup << 123 << endl;
    return 0;
}
```



7. Modul <format>

- U verziji jezika C++20 uvode se *moduli*, koji se koriste pomoću deklaracije: `import <naziv_modula>;`
- Modul <format> omogućava jednostavnije formatiranje izlaza, npr. za podatke

```
double r {1.5};  
double p {r * r * 3.14}; // 7.065
```

umesto dosadašnjeg načina formatiranja prikaza rezultata

```
cout << "Povrsina kruga r=" << r << " je "  
     << setprecision(2) // broj značajnih cifri  
     << p << "\n";      // 7.1
```

može se koristiti noviji način

```
cout << format("Povrsina kruga r={} je {:.2} \n", r, p);
```

8. Primeri programa

1. Lista prostih brojeva



8.1 Lista prostih brojeva

- Prosti brojevi
 - Generisanje i ispis zadanog broja n prostih brojeva
 - za generisanje liste prostih brojeva manjih od zadanog broja koristi se algoritam *Eratostenovo sito*
 - prosti brojevi se generišu u asocijativnom kontejneru tipa *set*
 - Spisak prostih brojeva upisuje se na tekstualni fajl **prosti.txt**



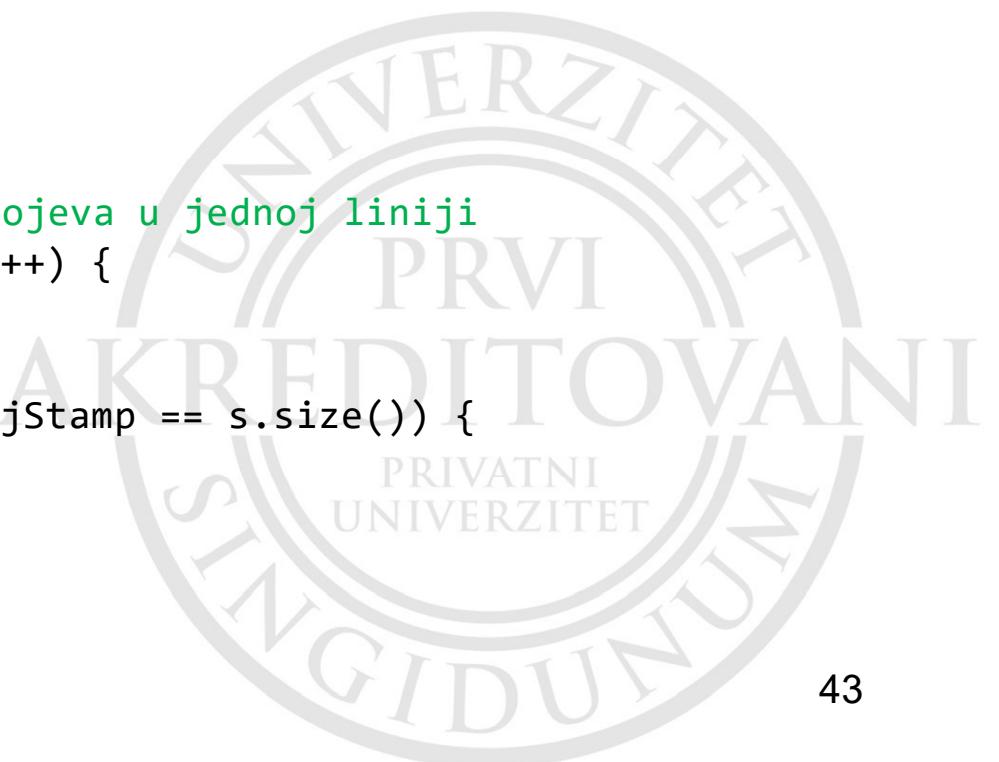
Prosti brojevi (1/3)

```
#include <iostream>
#include <fstream>
#include <set>
#include <string>

using namespace std;
typedef set<int>::iterator s_iter;

void stampaj_spisak(const set<int>& s) {
    const int BR_PO_LINIJI = 10;

    // Štampanje spiska prostih brojeva s
    int brojStamp = 0;      //brojač stampanih brojeva u jednoj liniji
    for (s_iter i = s.begin(); i != s.end(); i++) {
        cout << *i << " ";
        brojStamp++;
        if (brojStamp % BR_PO_LINIJI == 0 || brojStamp == s.size()) {
            cout << endl;
        }
    }
}
```



Prosti brojevi (2/3)

```
void sito(set<int>& s, int n) {
    // Kreiranje spiska celih brojeva 1..n
    for (int i = 1; i <= n; i++)
        s.insert(i);
    // Prosejavanje: brisanje brojeva koji nisu prosti
    for (int m = 2; m < *s.rbegin(); m++) {
        for (int k = m; k <= n; k++)
            s.erase(m*k); // brisanje iz skupa proizvoda dva broja
    }
    stampaj_spisak(s);
}

void sacuvaj_spisak(const set<int>& s, string putanja) {
    ofstream save_file(putanja); // konstruktor klase ostream
    if (!save_file) {
        cerr << endl << "Greska: fajl se ne moze otvoriti za upis!" << endl;
    } else {
        for (s_iter i = s.begin(); i != s.end(); i++)
            save_file << *i << " ";
        save_file << endl;
        save_file.close();
    }
}
```

Prosti brojevi (3/3)

```
int main() {
    int n;          // gornja granica intervala
    set<int> s;   // skup (prostih) brojeva

    cout << "Spisak svih prostih brojeva manjih od n\n"
        << "Unesite n: ";
    cin >> n;

    sito(s, n);

    sacuvaj_spisak (s, "prosti.txt");

    return 0;
}
```

```
Spisak svih prostih brojeva manjih od n
Unesite n: 50
1 2 3 5 7 11 13 17 19 23
29 31 37 41 43 47
```

prosti.txt

1 . 2 . 3 . 5 . 7 . 11 . 13 . 17 . 19 . 23 . 29 . 31 . 37 . 41 . 43 . 47 . CRLF

Napomena: oznaka kraja linije tekstualnog fajla je:

- Windows CRLF
- Linux/Unix LF
- Mac OS CR

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. O'Dwyer A., *Mastering the C++17 STL*, Packt, 2017
8. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
9. Web izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learnCPP.com/>
 - <http://www.stroustrup.com/>
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 13

Tehnike efikasnog programiranja u jeziku C++ i uvod u OO modelovanje

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Efikasnost poznatih algoritama
3. Metodologije razvoja softvera
4. Identifikacija klasa
5. Identifikacija veza između klasa
6. Primer projektovanja



1. Uvod

- Efikasnost programa
- Razvoj efikasnih algoritama
- Proces razvoja softvera



Efikasnost programa

- Efikasni programi razvijaju se na osnovu **efikasnih algoritama**
- Poređenje algoritama, npr. linearog i binarnog pretraživanja, može se izvršiti na osnovu testiranja, ali je zavisno od
 - hardverskih i softverskih osobina računara i trenutnih uslova konkurentnog izvršavanja
 - obima i osobina podataka, npr. da li su već sortirani
- Zbog toga se ocena efikasnosti algoritama vrši na osnovu njihovog asimptotskog ponašanja, funkcije rasta $O(n)$
- Postoje **opšti metodi** dizajniranja efikasnih algoritama, kao što su npr. dinamičko programiranje, podela problema na potprobleme (*divide-and-conquer*) i sistematsko pretraživanje (*backtracking*)

Razvoj efikasnih algoritama

- Dizajn algoritama predstavlja metod izgradnje matematičkog pristupa rešavanju problema, odnosno razvoju programa
- Analiza algoritama se bavi predviđanjem performansi algoritama
 - u praksi su identifikovani brojni obrasci (*design patterns*), šabloni metoda i načina upotrebe struktura podataka
npr. šablon **dekorater** (*decorator pattern*) omogućava dodavanje ponašanja nekom objektu bez uticaja na ponašanje drugih objekata iste klase
- Jedan od najvažnijih aspekata dizajna algoritama je razvoj algoritama male vremenske složenosti $O(n)$

Proces razvoja softvera

- Softverski sistemi se kreiraju da postoje određeno vreme
- Životni ciklus softverskog sistema (*system life cycle*) može se podeliti u dve osnovne faze:
 - fazu *razvoja* (završava *isporukom*)
 - fazu *rada* i održavanja (završava *zastarevanjem*)
- Zastarevanje pokreće razvoj/nabavku nove verzije sistema i povlačenje iz upotrebe stare verzije (*pensioning*)
- Proces razvoja softvera je pristup izgradnji, isporuci i održavanju softvera
 - parcijalno uređeni niz koraka usmerenih ka cilju
 - cilj je efikasna i predvidiva isporuka softverskog sistema, koji zadovoljava postavljene zahteve

2. Efikasnost poznatih algoritama

1. Primeri elementarnih algoritama
2. Vremenska složenost binarnog pretraživanja
3. Vremenska složenost sortiranja selekcijom
4. Pronalaženje Fibonačijevih brojeva (dinamičko programiranje)
5. Poređenje opštih funkcija rasta



2.1 Primeri elementarnih algoritama

- Vremenska složenost ugnježđenih petlji

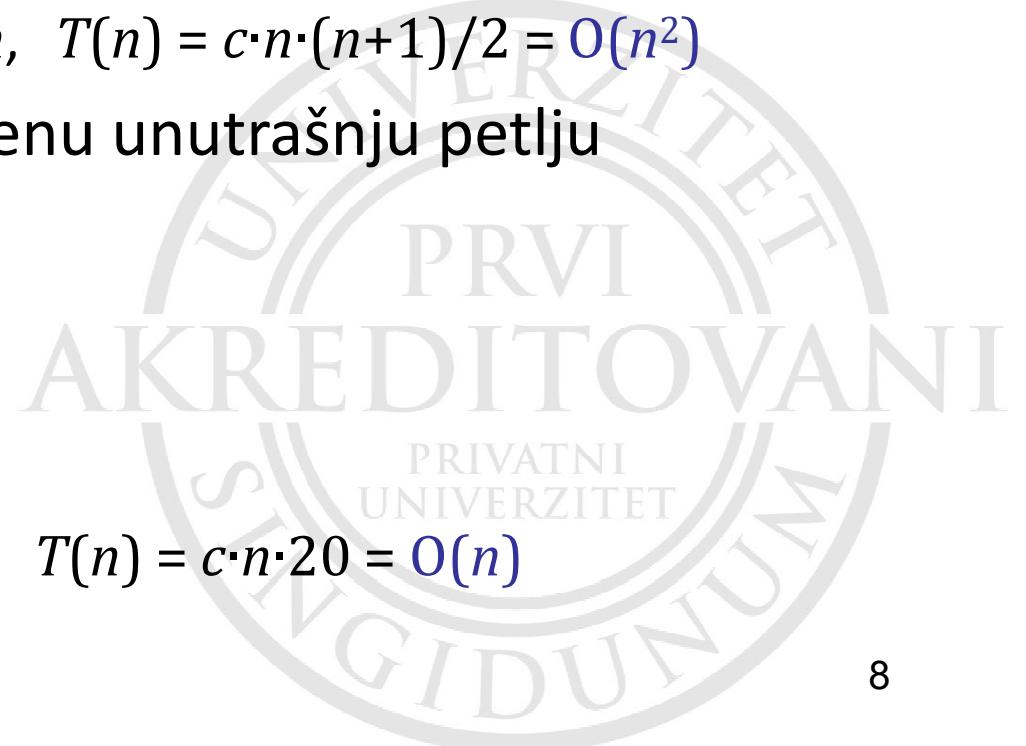
```
for (i=1; i<=n; i++) {  
    for (j=1; j<=i; j++) {  
        k = k + i + j;  
    }  
}
```

— spoljašnja n puta, unutrašnja $\sum_{1..n} n$, $T(n) = c \cdot n \cdot (n+1)/2 = O(n^2)$

- Vremenska složenost za izmenjenu unutrašnju petlju

```
for (i=1; i<=n; i++) {  
    for (j=1; j<=20; j++) {  
        k = k + i + j;  
    }  
}
```

— spoljašnja n puta, unutrašnja $20 \cdot n$, $T(n) = c \cdot n \cdot 20 = O(n)$



2.2 Vremenska složenost binarnog pretraživanja

- Binarno pretraživanje pronađe zadani element key u sortiranoj listi od n elemenata
 - jedna iteracija petlje izvrši se za konstantno vreme c
 - algoritam na svakom koraku petlje eliminiše $1/2$ elemenata, nakon dva poređenja
- Vremenska složenost algoritma je
$$\begin{aligned}T(n) &= T(n/2)+c = T(n/2^2)+c+c \\&= \dots = T(n/2^k)+k\cdot c = T(1)+c\cdot \log n \\&= 1+(\log n) + c = O(\log n)\end{aligned}$$
- Algoritam ima *logaritamsku* vremensku složenost

```
int binarySearch (const int list[],  
                  int key, int listSize) {  
    int low = 0;  
    int high = listSize - 1;  
  
    while (high >= low) {  
        int mid = (low + high)/2;  
        if (key < list[mid])  
            high = mid - 1;  
        else if (key == list[mid])  
            return mid; // pronađen  
        else  
            low = mid + 1;  
    }  
    return -low - 1; // nije pronađen  
}
```

2.3 Vremenska složenost sortiranja selekcijom

- Sortiranje selekcijom, počev od prvog elementa, pronalazi minimalni element u preostalom delu liste i po potrebi ga zameni s prvim
- Ponavlja postupak od narednog elementa liste
 - broj poređenja je $n-1$ u prvoj iteraciji, $n-2$ u drugoj itd.
- Ukupan broj operacija je
$$T(n) = (n-1) + c + (n-2) + c + \dots + 2 + c + 1 + c = (n-1)(n-1+1)/2 + c \cdot (n-1) = \frac{n^2}{2} - n/2 + c \cdot n - c = O(n^2)$$
- Algoritam ima *kvadratnu* vremansku složenost

```
void selectionSort(double list[], int lSize) {  
    for (int i = 0; i < lSize - 1; i++) {  
        // Pronaći min. vredn. u list[i..lSize-1]  
        double currMin = list[i];  
        int currMinInd = i;  
  
        for (int j = i + 1; j < lSize; j++) {  
            if (currMin > list[j]) {  
                currMin = list[j];  
                currMinInd = j;  
            }  
        }  
        // Ako je potrebno, zamena list[i] s  
        // list[currMinInd]  
        if (currMinInd != i) {  
            list[currMinInd] = list[i];  
            list[i] = currMin;  
        }  
    }  
}
```

2.4 Pronalaženje Fibonačijevih brojeva (dinamičko programiranje)

- Rekurzivna verzija

$$F(1) = 1, F(2) = 1$$

$$F(n) = F(n-1) + F(n-2)$$

ima složenost $O(2^n)$

- Veliki nedostatak algoritma su redundantna računanja, jer za svaki poziv gde je $n > 1$ generiše dva nova poziva, npr.

- za $\text{fib}(4)$, poziva $\text{fib}(3)$ i $\text{fib}(2)$
- za $\text{fib}(3)$, poziva (ponovo!) $\text{fib}(2)$ i $\text{fib}(1)$

```
// Računa Fibonačijev broj za
// zadani n
long fib(long n) {
    if (n == 0) // početak
        return 0;
    else if (n == 1) // početak
        return 1;
    else // rekurzivni poziv za n>1
        return fib(n-1) + fib(n-2);
```

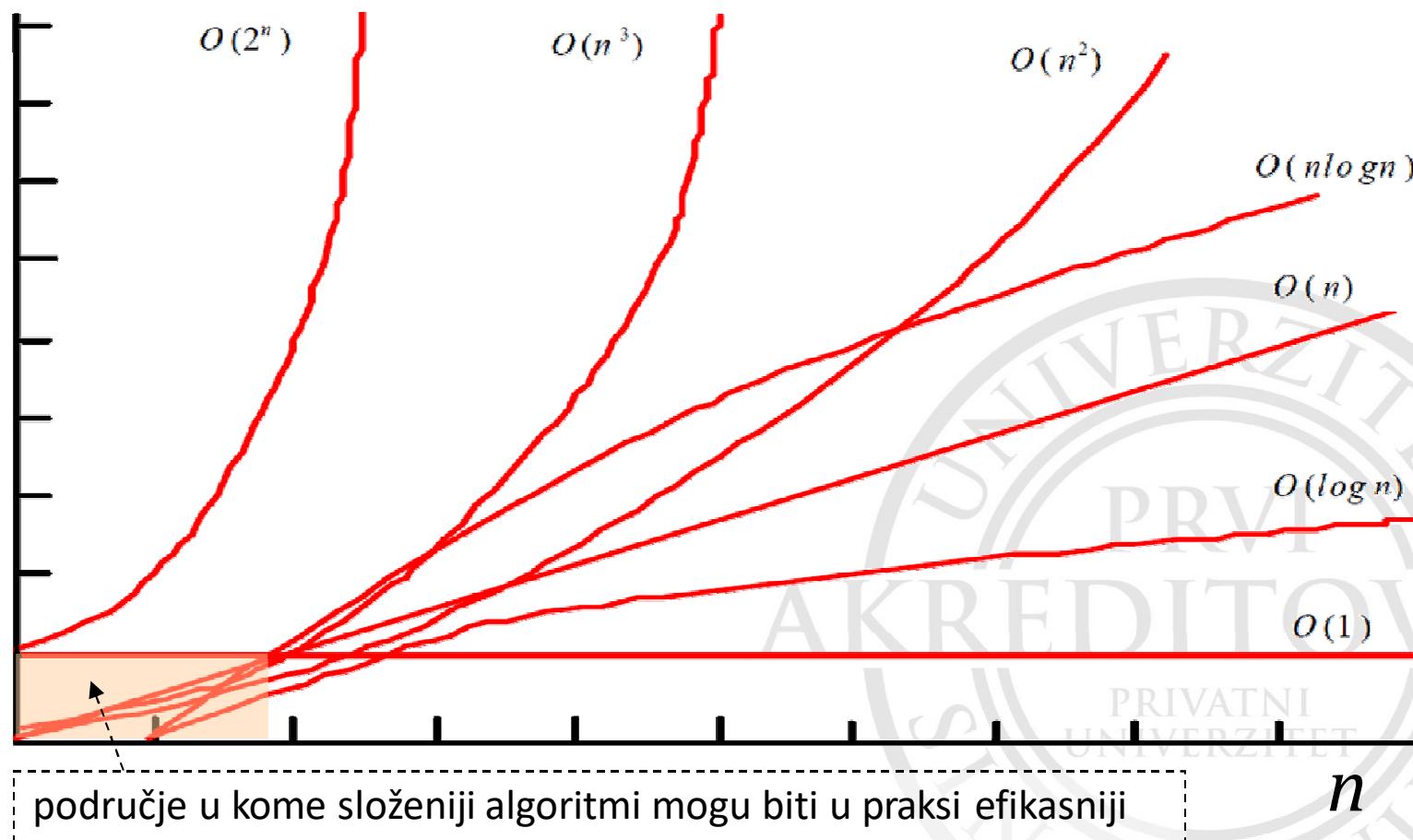
Pronalaženje Fibonačijevih brojeva (dinamičko programiranje)

- Bolje rešenje je algoritam *dinamičkog programiranja* složenosti $O(n)$, koji *ne ponavlja* računanja na svakom koraku

```
// Računa Fibonačijev broj za zadani n
long fib(long n) {
    long f0 = 0; // fib(0)
    long f1 = 1; // fib(1)
    long f2 = 1; // fib(2)
    if (n == 0)
        return f0; // početak
    else if (n == 1)
        return f1; // početak
    else if (n == 2)
        return f2;
    // n>1
    for (int i = 3; i <= n; i++) {
        f0 = f1;
        f1 = f2;
        f2 = f0 + f1;
    }
    return f2;
}
```

2.5 Poređenje opštih funkcija rasta

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < \dots$$



3. Metodologije razvoja softvera

1. Objektno orijentisani razvoj softvera
2. Metodologija razvoja
3. Sekvencijalna metodologija
4. Metodologija RUP
5. Ekstremno programiranje

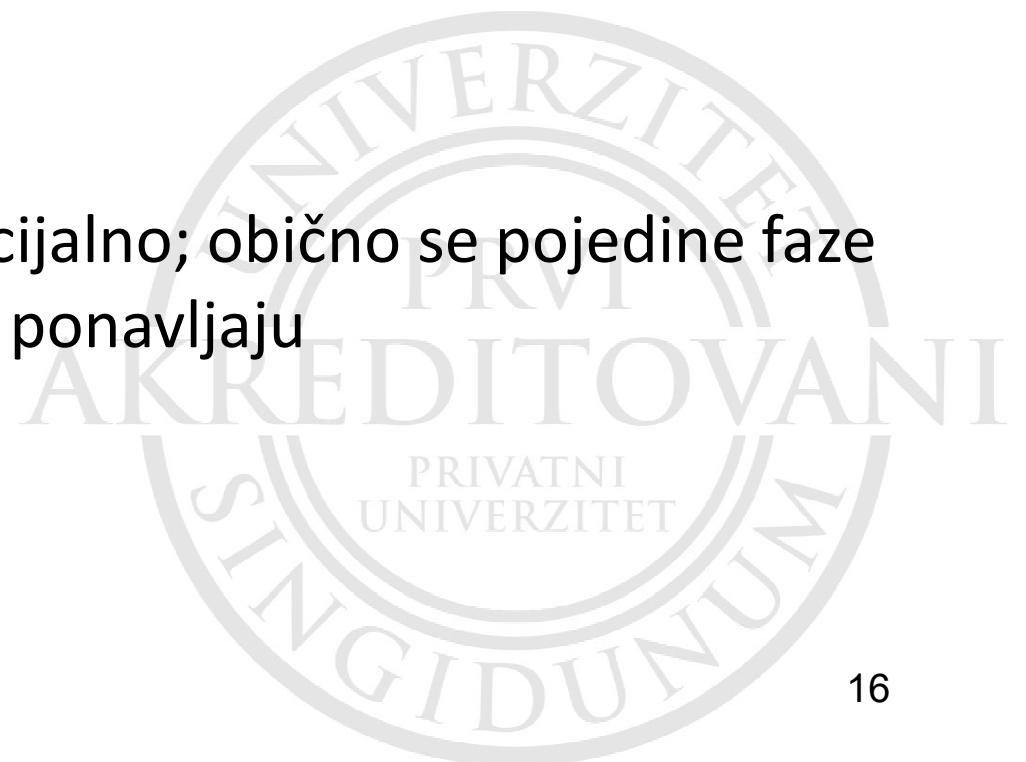


3.1 Objektno orijentisani razvoj softvera

- Objektno orijentisani pristup razvoju softvera, koji se koristi za složene softverske sisteme, obuhvata objektno orijentisani *analizu i projektovanje*
 - objektno orijentisana *analiza* je proces usmeren na ispitivanje problema i zahteva, a ne na njihovo rešavanje
npr. identifikuju se *objekti* od interesa, kao *Avion*, *Let* i *Pilot*
 - objektno orijentisano *projektovanje* je proces kreiranja konceptualnog rešenja koje zadovoljava postavljene zahteve, ali ne i njegovu implementaciju; u toku projektovanja definišu se *softverski objekti* i način njihove saradnje radi zadovoljenja zahteva
npr. klasa *Avion* ima atribut *registarskiBroj* i metod *getIstorijaLetenja()*

Faze razvoja softvera

- Razvoj softvera najčešće se posmatra kroz pet osnovnih *faza razvoja*:
 1. Analiza (*analysis*)
 2. Projektovanje (*design*)
 3. Implementacija (*implementation*)
 4. Testiranje (*testing*)
 5. Isporuka (*deployment*)
- Razvoj softvera ne teče sekvencialno; obično se pojedine faze razvoja delimično preklapaju ili ponavljaju



Projektovanje softvera

- Prilikom razvoja softvera kreira se *model sistema*, koji prikazuje softverski sistem s različitih aspekata, npr. posebnim vrstama UML dijagrama za
 - prikaz *strukture* softvera pomoću objekata, atributa, operacija i relacija (npr. dijagrami slučajeva korišćenja, klase i paketa)
 - prikaz *ponašanja* softvera kroz prikaz saradnje objekata i promena njihovih unutrašnjih stanja (npr. dijagrami sekvenci, aktivnosti, stanja)
- Standard UML definiše 14 dijagrama u tri kategorije
- Softverski alati za projektovanje softvera omogućavaju kreiranje UML modela sistema i generisanje koda na osnovu UML dijagrama (samo strukture koda ili celog sistema)
 - npr. Microsoft *Visual Studio* i IBM *Rhapsody*

3.2 Metodologija razvoja

- Metodologija razvoja (*system development methodology*) je je skup aktivnosti, metoda, iskustava, preporuka i automatizovanih alata koji se koriste za razvoj i neprekidno usavršavanje softvera
- Postoje različite **klasične** (sekvencijalna), **novije** (iterativna, inkrementalna, kombinovane) i **agilne** metodologije
- Agilne metodologije
 - manje stroge metodologije, koje manje projekte i timove ne opterećuju birokratijom, najpoznatije su su XP (*Extreme Programming*) i Scrum
 - podrazumevaju adaptivno planiranje, evolutivni razvoj i inkrementalnu isporuku softvera

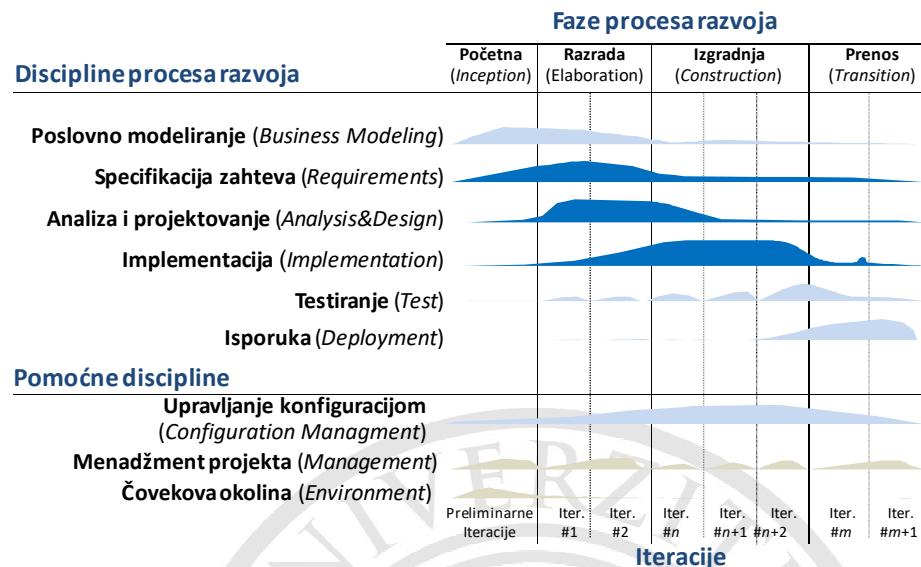
3.3 Sekvencijalna metodologija

- **Sekvencijalni razvoj** ili model vodopada (*waterfall*) je pristup razvoju gde se svaka faza razvoja završi u potpunosti, nakon čega se prelazi na sledeću fazu
- Metodologija nije adekvatna za sisteme velike kompleksnosti
 - sve veći obim aplikacija
 - veliki ili distribuirani timovi
 - povećana je tehnička složenost sistema
 - stalne novine u tehnologijama
 - produžava trajanje projekta
- Glavni problem ovog pristupa je što ne omogućava identifikovanje i umanjenje *rizika* u ranim fazama projekta



3.4 Metodologija RUP

- Konkretna objektno orijentisana metodologija *Rational Unified Process* koju je uvela kompanija *Rational* (kasnije IBM)
 - iterativno-inkrementalna
 - sve faze razvoja se realizuju kroz niz iteracija i inkremenata
 - cilj je što kvalitetniji rezultat u posmatranom vremenu
 - *na kraju svake iteracije je kod*
 - metodologija ima sopstvene softverske alate



3.5 Ekstremno programiranje

- Ekstremno programiranje (XP) je *agilna* metodologija, koja podrazumeva i prihvata stalne promene sistema kao činjenicu
- Realizuje se kroz male ili srednje razvojne timove, koji rade u tesnoj *saradnji s korisnikom*
- Ne koristi se precizno planiranje, već se teži brzim, opipljivim rezultatima, koji se odmah predočavaju korisniku
- Elementi metodologije su specifične preporuke (*practices*)
 - realno planiranje, razvoj u malim koracima
 - programiranje u parovima (na smenu)
 - korisnik je dostupan članovima tima *celo vreme na lokaciji*
 - testiranje je *kontinualno* (vrše i programeri i korisnici)
 - kodiranje se vrši prema standardima (samodokumentujući kod)

4. Identifikacija klasa

- Proces otkrivanja klasa
- Svojstva dobrog OO modela



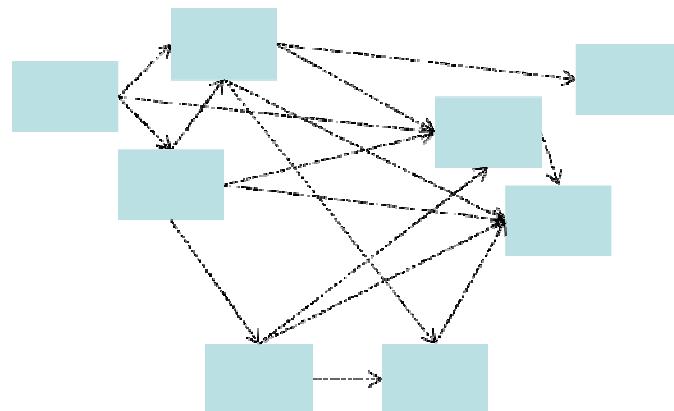
Proces otkrivanja klasa

- Osnovni problem objektno orijentisanog projektovanja je otkrivanje *entiteta* i ustanavljanje njihovih *svojstava* i *zadataka*
 - koji će se predstaviti u obliku *klasa* i njihovih *atributa* i *metoda*
- Popularni naziv jedne neformalne tehnike evidentiranja klasa su CRC kartice (*Classes-Responsibilities-Collaborators*)
- Na osnovu *anализе проблема*, iz dokumentacije, upitnika, intervjuja, posmatranjem i sl., uočavaju se **klase**, njihovi **atributi** i **veze** s drugim klasama
 - tehnika: imenice, glagoli, pridevi iz opisa problema

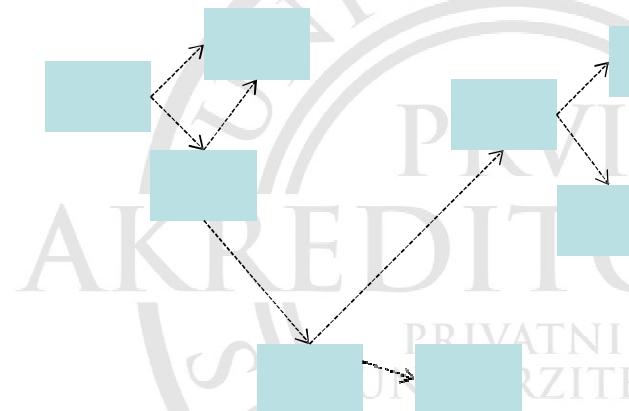
Zaduženja	Klase	Sarađuju
Račun		
izračunati ukupno zaduženje	StavkaRacuna	
...		

Svojstva dobrog objektno orijentisanog modela

- Unutrašnja kohezija klasa (*cohesion*)
 - klasa treba da predstavlja jedan pojam iz domena problema, npr. iako je oblik točka automobila krug, točak nije geometrijski objekt
 - sva svojstva klase treba da se odnose na pojam koji klasa predstavlja
- Slaba međusobna povezanost klasa (*coupling*)
 - minimizacija veza između klasa (npr. zavisnosti)



Jaka povezanost klasa
(*high coupling*)



Slaba povezanost klasa
(*low coupling*)

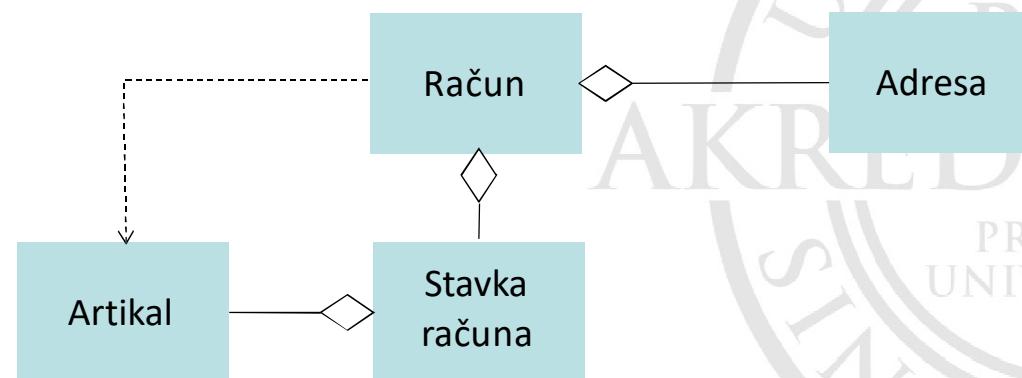
5. Identifikacija veza između objekata

- Vrste veza
- Prikaz veza u UML dijagramu klasa
- Implementacija agregacije



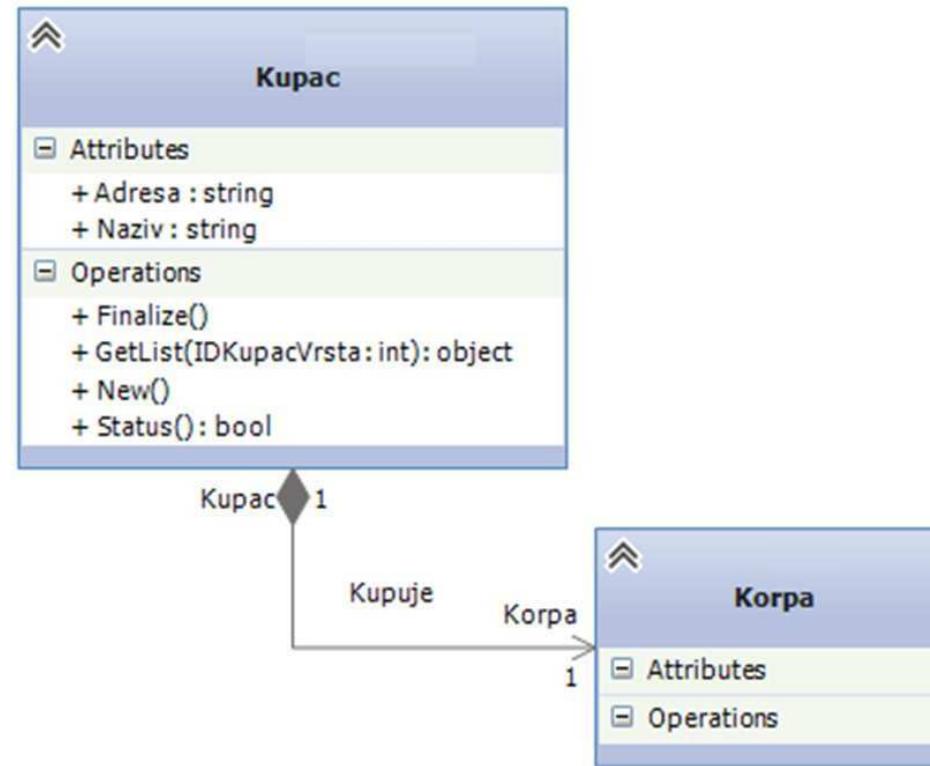
Vrste veza

- Veza zavisnosti (*uses*, "koristi")
 - klasa je povezana s drugom klasom, ako neka njena funkcija član na neki način *koristi* objekt druge klase
- Veza nasleđivanja (*is-a*, "jeste")
 - klasa je *specijalni slučaj* druge klase, nasleđuje njeni svojstva
- Veza agregacije ili kompozicije (*has-a*, "ima")
 - klasa *sadrži* druge klase, koje su njeni delovi



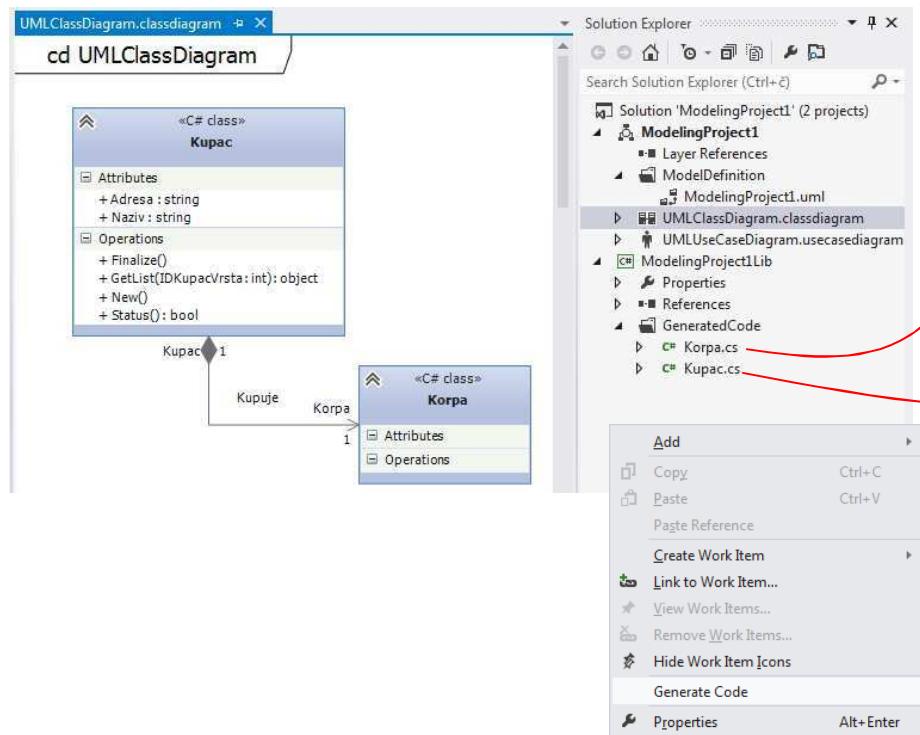
Prikaz veza u UML dijagramu klasa

- Dijagram klasa je osnovni projektni dijagram za opis *strukture sistema*
- Modeli objekata i veze između objekata
 - klase (iz *Use Case* i dr.)
 - atributi (vidljivost, tip)
 - metodi (vidljivost, parametri)
 - veze (*multiplikativnost*)
- Dijagrami se razvijaju postepeno. U svakoj fazi je moguće *generisati kod*



Primer: Generisanje kôda na osnovu dijagrama klasa (Visual Studio)

- Primer: UML definicija klasa Kupac i Korpa
 - Generisana struktura koda:



The diagram illustrates a code migration or comparison between two versions of a class definition. The left snippet shows the original state, and the right snippet shows the target state after changes.

Left Snippet (Original State):

```
1 // ...
2 // <auto-generated>
3 //   This code was generated by a tool
4 //   Changes to this file will be lost if the code is regenerated.
5 // </auto-generated>
6 //...
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11
12 public class Kupac
13 {
14     public virtual string Naziv
15     {
16         get;
17         set;
18     }
19
20     public virtual string Adresa
21     {
22         get;
23         set;
24     }
25
26     public virtual Korpa Korpa
27     {
28         get;
29         set;
30     }
31
32     public virtual object GetList(int IDKupacVrsta)
33     {
34         throw new System.NotImplementedException();
35     }
36
37     public virtual void New()
38     {
39         throw new System.NotImplementedException();
40     }
41
42     public virtual bool Status()
43     {
44         throw new System.NotImplementedException();
45     }
46
47     public virtual void Finalize()
48     {
49         throw new System.NotImplementedException();
50     }
51 }
52 }
```

Right Snippet (Target State):

```
1 // ...
2 // <auto-generated>
3 //   This code was generated by a tool
4 //   Changes to this file will be lost
5 // </auto-generated>
6 //...
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11
12 public class Kupac
13 {
14     public virtual string Naziv
15     {
16         get;
17         set;
18     }
19
20     public virtual string Adresa
21     {
22         get;
23         set;
24     }
25
26     public virtual Korpa Korpa
27     {
28         get;
29         set;
30     }
31
32     public virtual object GetList(int IDKupacVrsta)
33     {
34         throw new System.NotImplementedException();
35     }
36
37     public virtual void New()
38     {
39         throw new System.NotImplementedException();
40     }
41
42     public virtual bool Status()
43     {
44         throw new System.NotImplementedException();
45     }
46
47     public virtual void Finalize()
48     {
49         throw new System.NotImplementedException();
50     }
51 }
52 }
```

Red arrows point from the original code in the left snippet to the corresponding modified code in the right snippet, specifically highlighting the removal of the regenerate warning and the addition of the final line in the right snippet.

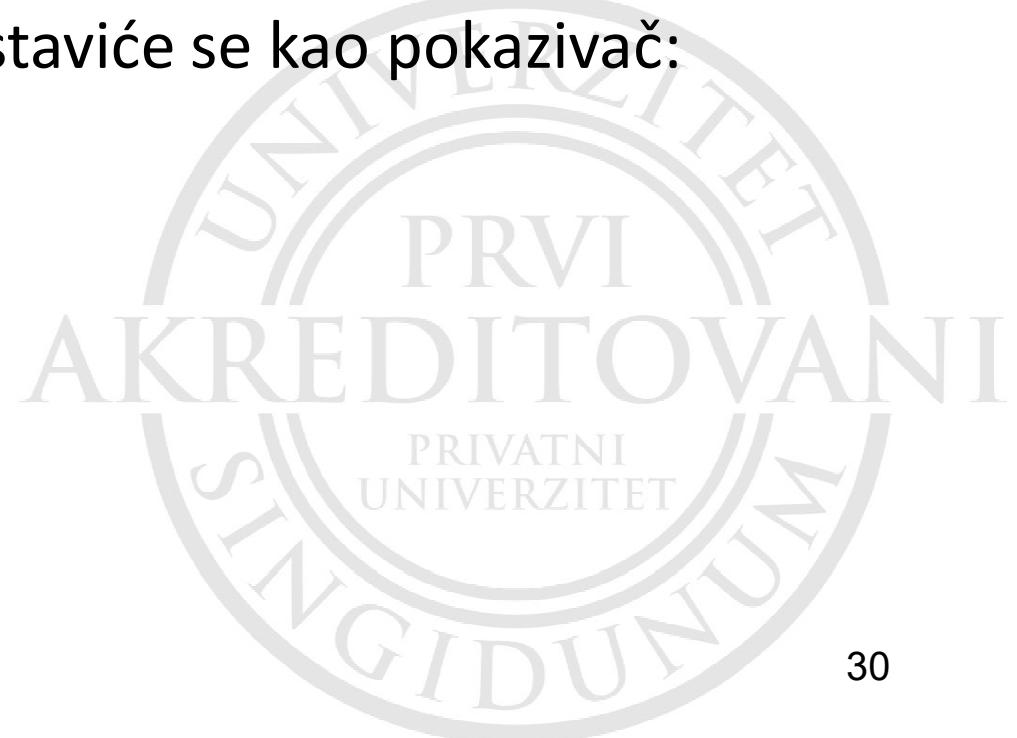
Implementacija agregacije

- Agregacija se implementira kao *član podatak*, koji može biti promenljiva, vektor ili pokazivač
- Način implementacije zavisi od multiplikativnosti agregacije:
 - $1 : 1$ - npr. svaki tekući račun ima jednog vlasnika
 - $1 : 0..1$ - npr. svaki departman hotela ima 0 ili 1 recepcionera
 - $1 : *$ - npr. svaka kompanija ima više zaposlenih
- Objekt tipa *vektor* može se koristiti za prikaz veze 1:više
- Umesto objekata koriste se *pokazivači*:
 - za veze $1 : 0..1$
 - za polimorfne klase, gde povezuju objekt s drugim objektom, koji pripada osnovnoj ili izvedenoj klasi
 - za deljenje objekata (*object sharing*)

Ilustracija: Bankarski račun

- Klasa **BankarskiRacun** sadrži objekt klase **Osoba**, koja je vlasnik računa i može se u programu predstaviti kao
 - objekt klase **Osoba** ili
 - pokazivač **Osoba***
- Pošto osoba može da ima više različitih bankarskih računa, koji *dele* podatke o osobi, predstaviće se kao pokazivač:

```
class BankarskiRacun {  
    ...  
    private:  
        Osoba* vlasnik;  
};
```



Ilustracija: Automobil

- Klasa **Automobil** povezana je s klasom **Tocak**
- Pošto automobil ima *više* točkova, mogu se predstaviti
 - vektorom objekata klase **Tocak** ili
 - pokazivačem na objekt klase **Tocak***
- Pošto točak može biti deo samo *jednog* automobila, predstaviće se kao objekt

```
class Automobil {  
    ...  
private:  
    vector<Tocak> tockovi;  
};
```



6. Primer projektovanja

- Koraci projektovanja dela Veb aplikacije



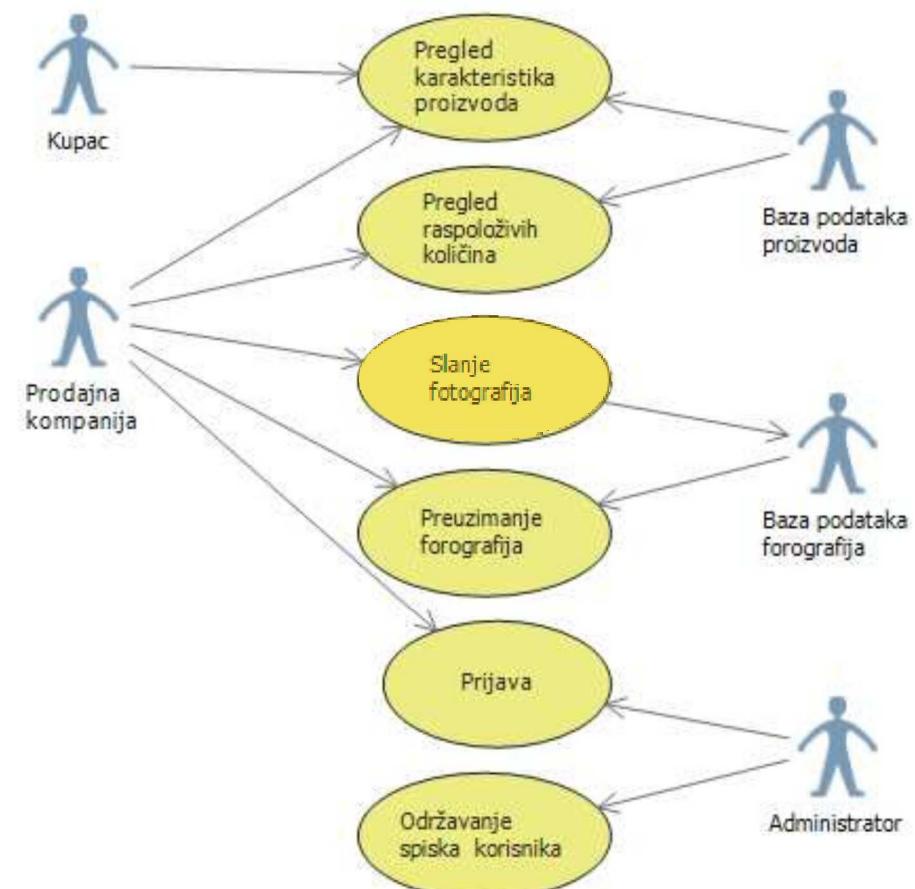
Koraci projektovanja dela Veb aplikacije

1. Specifikacija korisničkih zahteva (*Software Requirements*)
 - deo specifikacije su UML dijagrami slučajeva korišćenja
2. Razrada slučajeva korišćenja (*Use Cases*)
 - ponašanje pomoću UML dijagrama ponašanja, npr. dijagrama sekvenci
 - struktura pomoću UML dijagrama strukture, npr. dijagrama klasa
3. Generisanje koda, uglavnom na osnovu dijagrama klasa



Specifikacija korisničkih zahteva aplikacije (*Software Requirements*)

- Korisnički zahtevi se analiziraju kroz model slučajeva korišćenja (*Use Case*)
 - učesnici (akteri)
 - slučajevi korišćenja
 - veze između njih
- Dijagrami i njihove veze su samo pomoćna sredstva
 - *slučajevi korišćenja su tekstualni dokumenti - njihova izrada je pisanje teksta*



Početni diagram sekvenci slučaja

Slanje fotografija (upload)

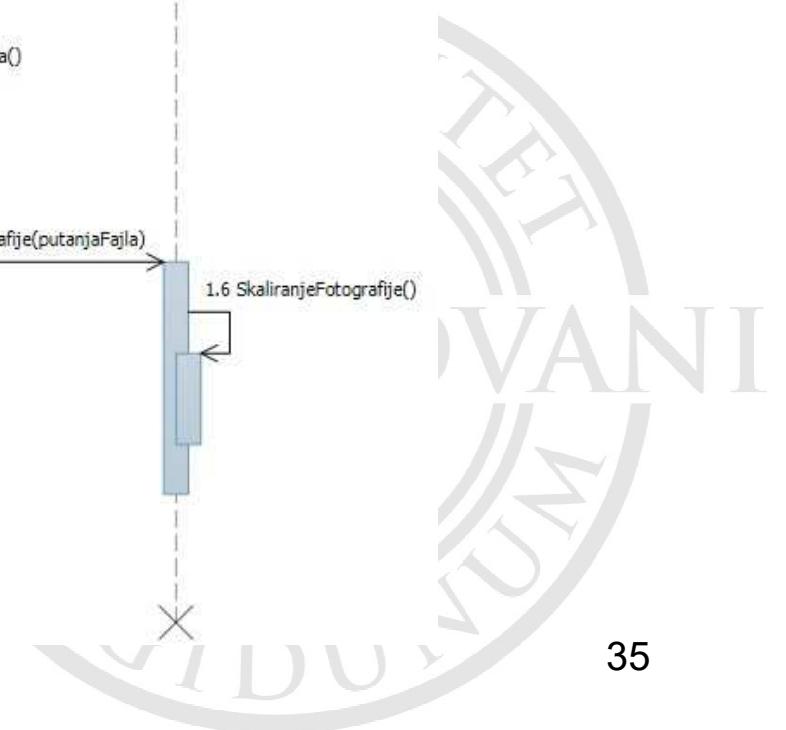
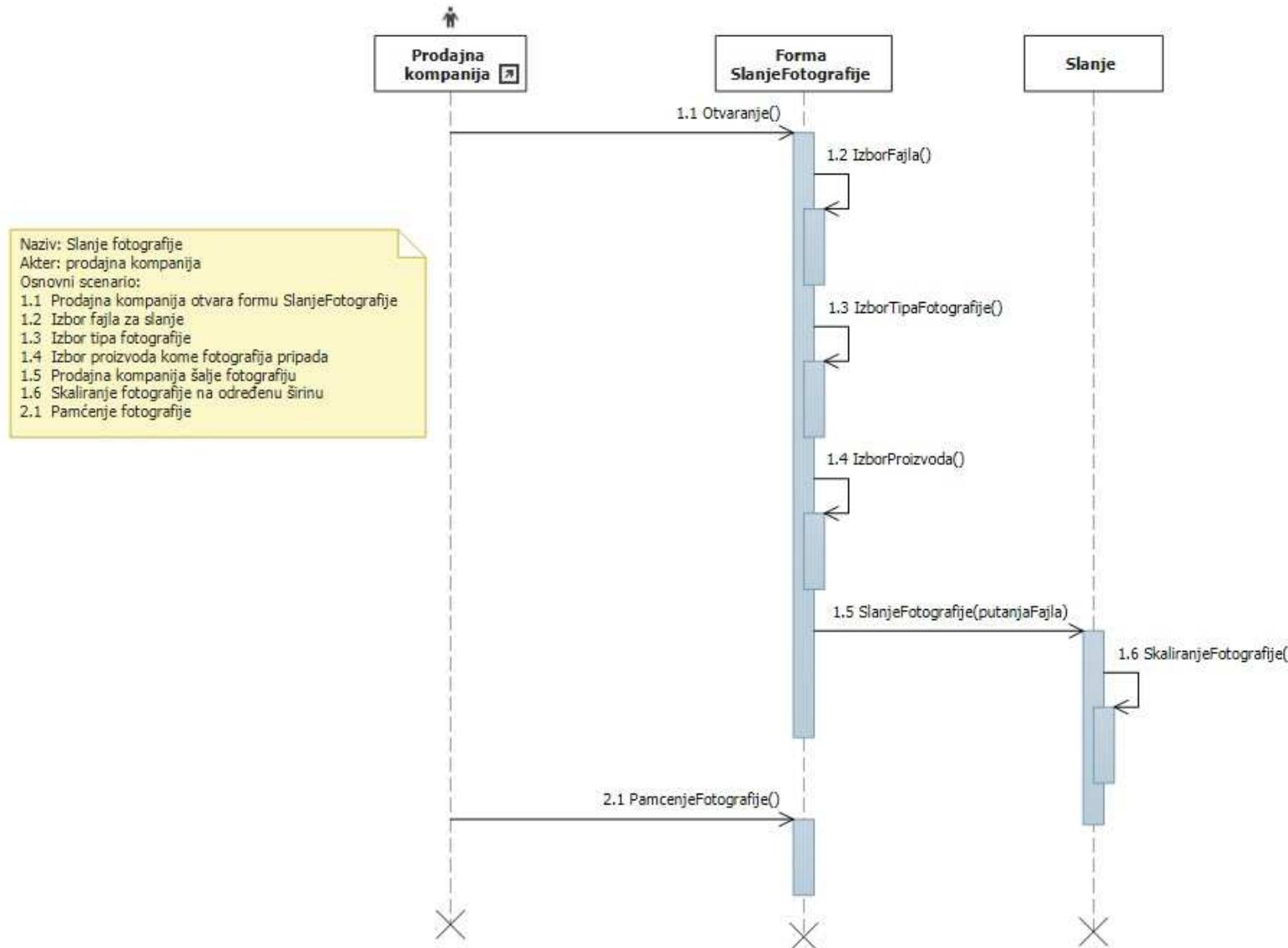
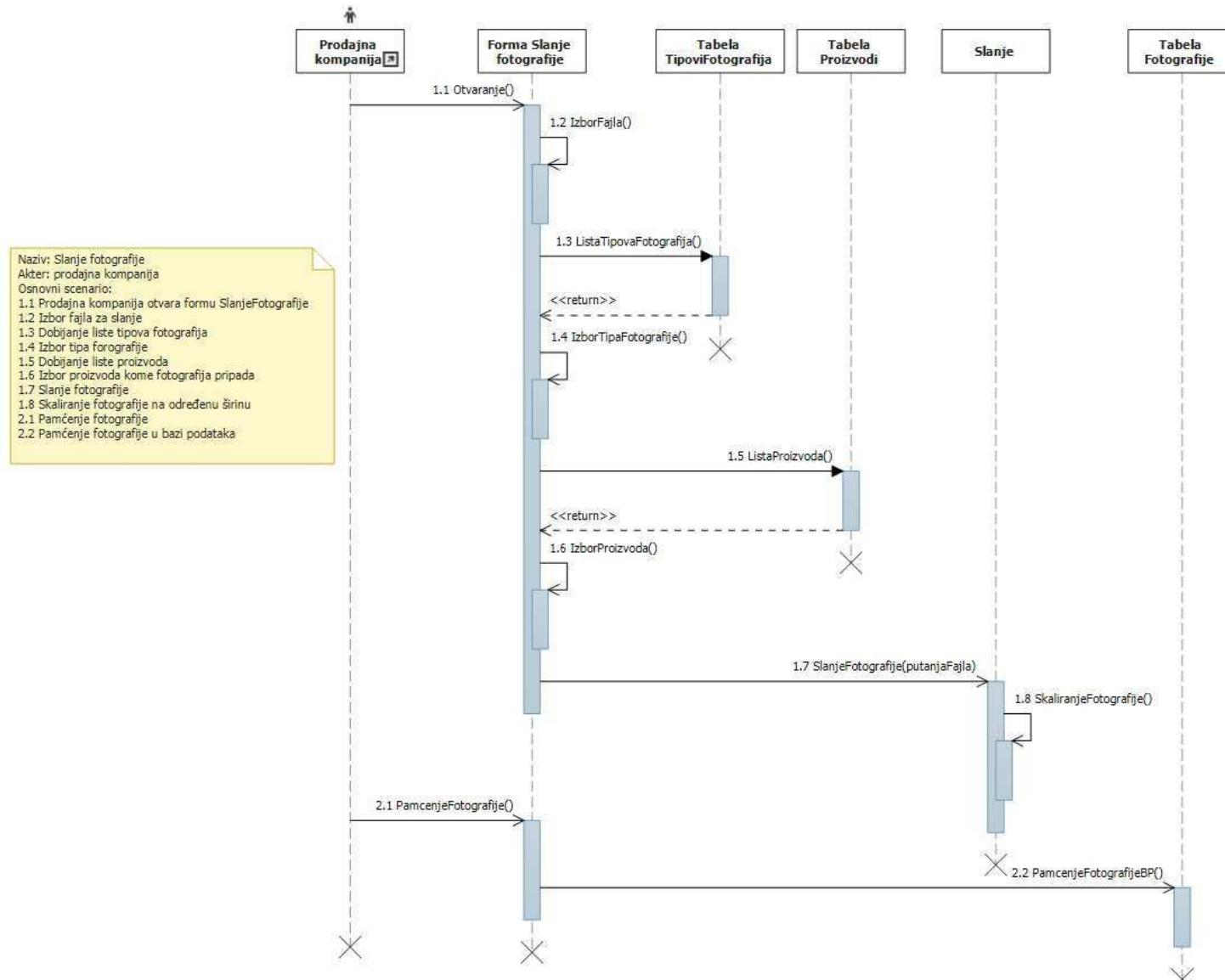


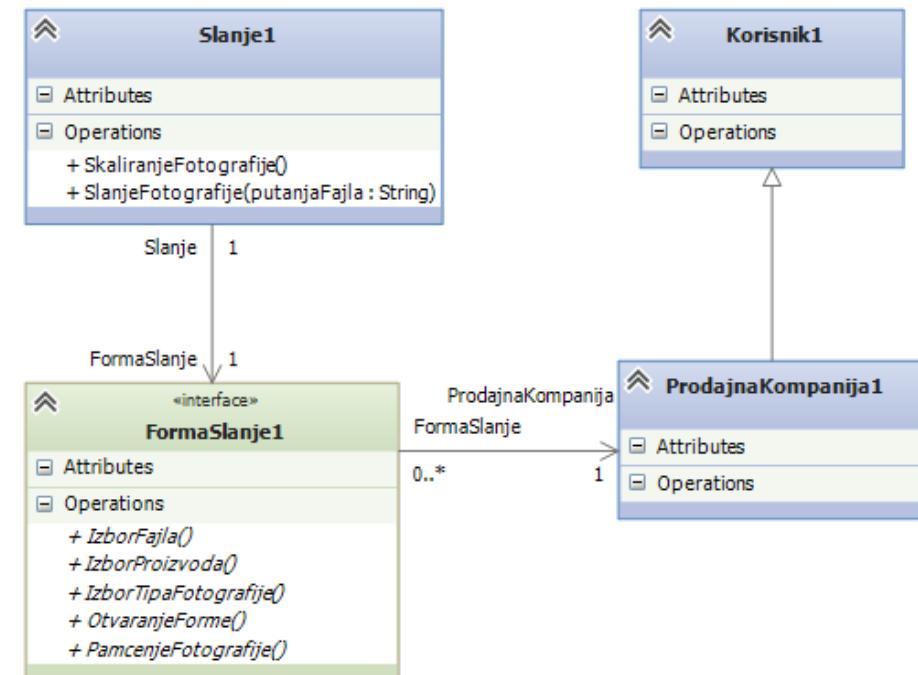
Diagram sekvenci slučaja

Slanje fotografija (upload)

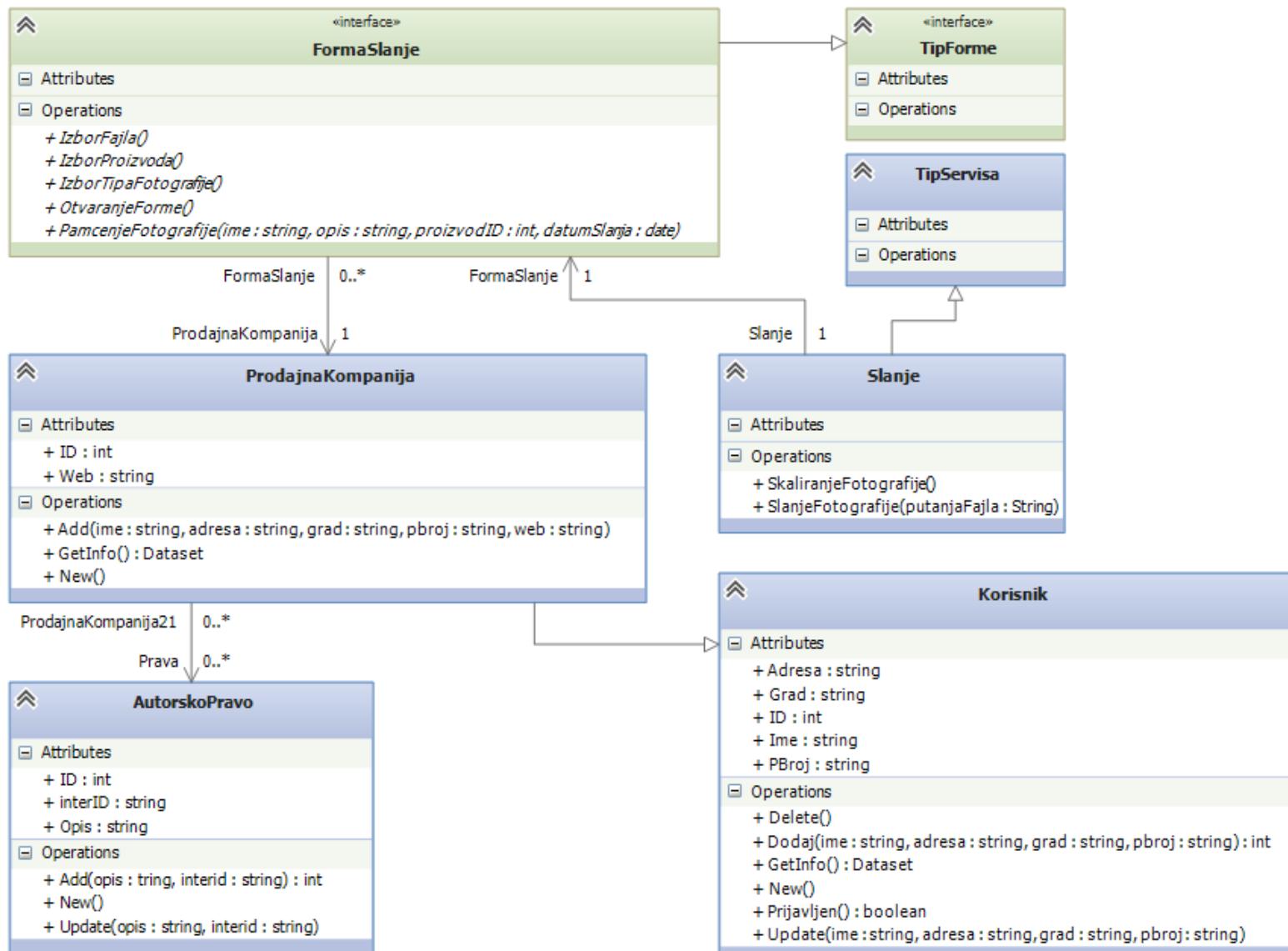


Dijagram klasa (početni)

- Klase s trenutno poznatim elementima
 - Slanje (upload)
 - FormaSlanje
 - Korisnik
 - ProdajnaKompanija



Dijagram klasa (konačna verzija)

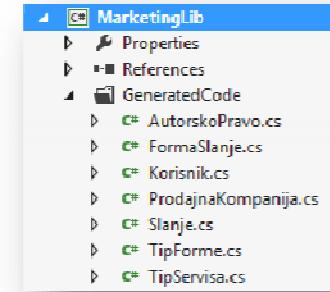


Struktura generisanog koda (jezik se posebno bira: za Veb aplikaciju C#)

```

1 //-----
2 // <auto-generated>
3 //   This code was generated by a tool
4 //   Changes to this file will be lost if the code is regenerated.
5 // </auto-generated>
6 //-----
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11
12 public class AutorskoPravo
13 {
14     public virtual int ID
15     {
16         get;
17         set;
18     }
19
20     public virtual string Opis
21     {
22         get;
23         set;
24     }
25
26     public virtual string interID
27     {
28         get;
29         set;
30     }
31
32     public virtual void New()
33     {
34         throw new System.NotImplementedException();
35     }
36
37     public virtual int Add(string opis, string interid)
38     {
39         throw new System.NotImplementedException();
40     }
41
42     public virtual void Update(string opis, string interid)
43     {
44         throw new System.NotImplementedException();
45     }
46
47 }

```

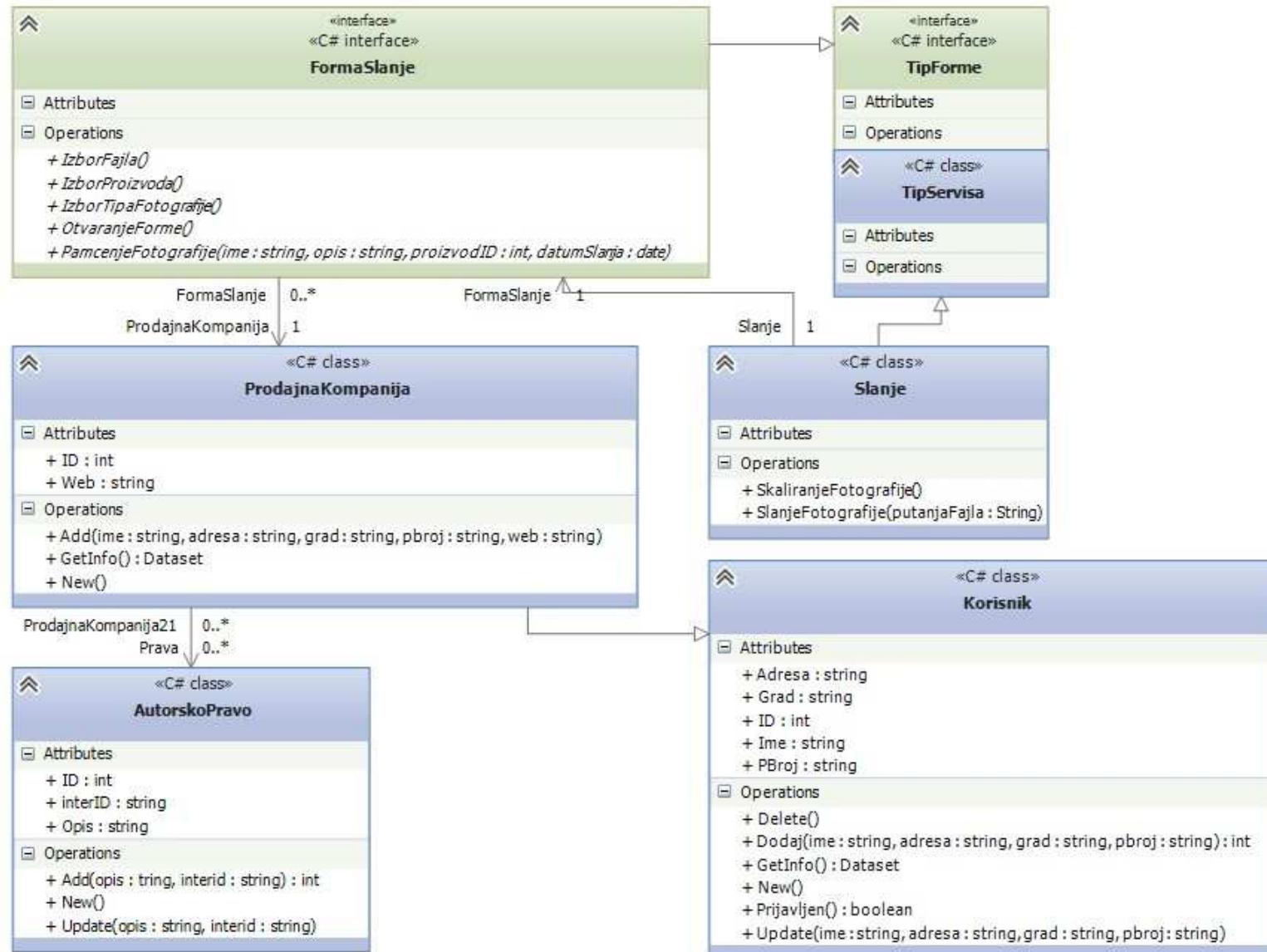


```

Slanje
1 //-----
2 // <auto-generated>
3 //   This code was generated by a tool
4 //   Changes to this file will be lost if the code is regenerated.
5 // </auto-generated>
6 //-----
7 using System;
8 using System.Collections.Generic;
9 using System.Linq;
10 using System.Text;
11
12 public class Slanje : TipServisa
13 {
14     public virtual FormaSlanje FormaSlanje
15     {
16         get;
17         set;
18     }
19
20     public virtual void SlanjeFotografije(string putanjaFajla)
21     {
22         throw new System.NotImplementedException();
23     }
24
25     public virtual void SkaliranjeFotografije()
26     {
27         throw new System.NotImplementedException();
28     }
29
30 }
31

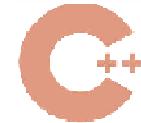
```

Dijagram klasa nakon generisanja koda



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Liang D., *Introduction to Programming With C++*, Pearson Education, 2014
3. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
4. Deitel P, Deitel H., *C++ How to Program*, 9th Ed, Pearson Education, 2014
5. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
6. Horton I., *Beginning C++*, Apress, 2014
7. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 14

Algoritmi pretraživanja iz biblioteke šablonu jezika C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Pretraživanja niza objekata metodima biblioteke STL
3. Particija niza objekata metodima biblioteke STL
4. Algoritmi binarnog pretraživanja biblioteke STL
5. Primer: Implementacija heš tabela



1. Uvod

- Problem pretraživanja
- Algoritmi pretraživanja biblioteke STL



Problem pretraživanja

- Pretraživanje skupova objekata u memoriji prema sadržaju ili delu njihovog sadržaja opšti je problem programiranja
- **Asocijativni** kontejneri omogućavaju pronalaženje elemenata po sadržaju za vreme $O(\log(n))$ ili $O(1)$
- **Kontejneri sekvenci** su linearne strukture (nizovi), u kojima se načelno objekti čija pozicija u nizu nije unapred poznata pronalaze *pretraživanjem* za vreme $O(n)$
- Pronalaženje sadržaja u *sortiranim* nizovima objekata može se izvršiti za vreme $O(\log(n))$
 - osim toga, sortiranje znatno ubrzava *ažuriranje* nizova objekata
- U biblioteci STL postoji algoritmi pretraživanja sekvenci (nizova) objekata na različite načine

Algoritmi pretraživanja biblioteke STL

- Biblioteka STL nudi tri algoritma za pronalaženje objekta u nizu definisanom pomoću prva dva argumenta (iteratora)
 - `find()` - pronalazi prvi objekt jednak trećem argumentu
 - `find_if()` - pronalazi prvi objekt za koji je *istinit predikat* naveden u trećem argumentu (vraća vrednost *true*); predikat ne sme da menja objekt koji ispituje i može se zadati kao *lambda izraz*: anonimna funkcija, koja se definiše na mestu njene upotrebe, npr.
`[element](parametri) {return Logički izraz;}`
 - `find_if_not()` - pronalazi prvi objekt za koji *nije istinit* predikat naveden u trećem argumentu (vraća vrednost *false*); predikat ne sme da menja objekt koji ispituje i može se zadati kao *lambda izraz*
- Svaki od algoritama vraća *iterator* koji pokazuje na pronađeni objekt ili na kraj niza, ako objekt nije pronađen

2. Pretraživanja niza objekata metodima biblioteke STL

1. Pronalaženje elementa u nizu objekata
2. Pronalaženje nekog od elemenata niza u nizu objekata
3. Pronalaženje ponavljanja elemenata drugog niza u nizu objekata



2.1 Pronalaženje elementa u nizu objekata

- Pronalaženje elementa u nizu može se izvršiti pomoću algoritma `find()`, npr.

```
std::vector<int> brojevi {5,46,-5,-6,23,17,5,9,6,5};  
int broj {23};  
auto iter = std::find(std::begin(brojevi),  
                      std::end(brojevi),  
                      broj);  
  
if (iter != std::end(brojevi))  
    std::cout << broj << " je pronađen\n";
```

- Element je pronađen ako iterator `end()` pokazuje na element niza, inače pokazuje na kraj, iza poslednjeg elementa niza
- Pronalaženje više instanci elementa u zadatom nizu vrši se ponavljanjem pretraživanja, od sledećeg elementa do kraja

Primer: Pronalaženje svih pojava elementa niza

```
#include <iostream>
#include <vector>
#include <algorithm>

int main() {
    std::vector<int> brojevi { 5, 46, -5, -6, 23, 17, 5, 9, 6, 5 };
    size_t n{};
    int broj{ 5 };      // broj 5 se nalazi 3 puta u nizu brojevi
    auto poc_it = std::begin(brojevi);
    auto kraj_it = std::end(brojevi);
    while ((poc_it = std::find(poc_it, kraj_it, broj)) != kraj_it) {
        ++n;
        ++poc_it;
    }
    std::cout << "Element " << broj << " je pronadjen " << n << " puta"
           << std::endl;
    return 0;
}
```

Element 5 je pronadjen 3 puta

Pronalaženje prvog većeg elementa u nizu

- Pronalaženje elementa u nizu koji je veći od neke vrednosti može se izvršiti algoritmom `find_if()`, npr.

```
std::vector<int>brojevi {5,46,-5,-6,23,17,5,9,6,5};  
int broj {5}; // prvi veći element niza je 46  
auto iter = std::find_if(std::begin(brojevi),  
                         std::end(brojevi),  
                         [broj](int n) { return n > broj; });  
if (iter != std::end(brojevi))  
    std::cout << "Broj " << *iter << " je veći od " << broj;
```

- Treći argument algoritma `find_if` je anonimni *lambda izraz* koji definiše *predikat*, koji se izračunava za svaki element niza
- Pronalaženje više svih elemenata većih od zadanog vrši se ponavljanjem pretraživanja, od sledećeg elementa do kraja

Pronalaženje svih elemenata u nizu koji nisu veći od zadatog elementa

- Algoritam `find_if_not()` pogodan je za pronalaženje elemenata u nizu koji *nisu veći* od zadatog, odnosno koji su *manji ili jednaki* zadatom, npr.

```
std::vector<int> brojevi {5,46,-5,-6,23,17,5,9,6,5};  
size_t n {};  
int broj {5}; // ima 5 elemenata koji nisu veći od 5  
auto poc_iter = std::begin(brojevi);  
auto kraj_iter = std::end(brojevi);  
while ((poc_iter = std::find_if_not(poc_iter, kraj_iter,  
[broj](int n){return n>broj;}) )  
!= kraj_iter){  
    ++n;  
    ++poc_iter;  
}  
std::cout << n << " elemenata nije veće od "<< broj << std::endl;
```

2.2 Pronalaženje nekog od elemenata niza u nizu objekata

- Pronalaženje u zadanom nizu prve pojave nekog elementa iz drugog niza, može se izvršiti algoritmom `find_first_of()`
 - poređenje se vrši standardnim ili preklopljenim operatorom `==` (za korisničke klase)
- Npr. prvi samoglasnik u zadanom tekstu pronalazi segment

```
std::string tekst {"Tekst za pretrazivanje"}; // prvi je 'e'  
std::string samoglasnici {"aeiou"};  
auto iter = std::find_first_of(std::begin(tekst),  
                               std::end(tekst),  
                               std::begin(samoglasnici),  
                               std::end(samoglasnici));  
  
if (iter != std::end(tekst))  
    std::cout << "Pronadjen samoglasnik '" << *iter << "'"  
           << std::endl;
```

Primer: Pronalaženje svih elemenata nekog niza objekata u drugom nizu

```
#include <iostream>
#include <string>
#include <algorithm>

int main() {
    std::string tekst {"Tekst za pretrazivanje"};
    std::string samoglasnici {"aeiou"};
    std::string pronadjeni {}; // zapisuje sve pronađene znakove
    for(auto iter = std::begin(tekst);
        (iter = std::find_first_of(
            iter,
            std::end(tekst),
            std::begin(samoglasnici),
            std::end(samoglasnici))) != std::end(tekst); )
        pronadjeni += *(iter++);
    std::cout << "Pronadjeni samogl. '" << pronadjeni << "'"
        << std::endl;
    return 0;
}
```

Pronadjeni samogl. 'eaeaiae'

Pronalaženje u nizu objekata nekog od elemenata niza za koji važi zadani predikat

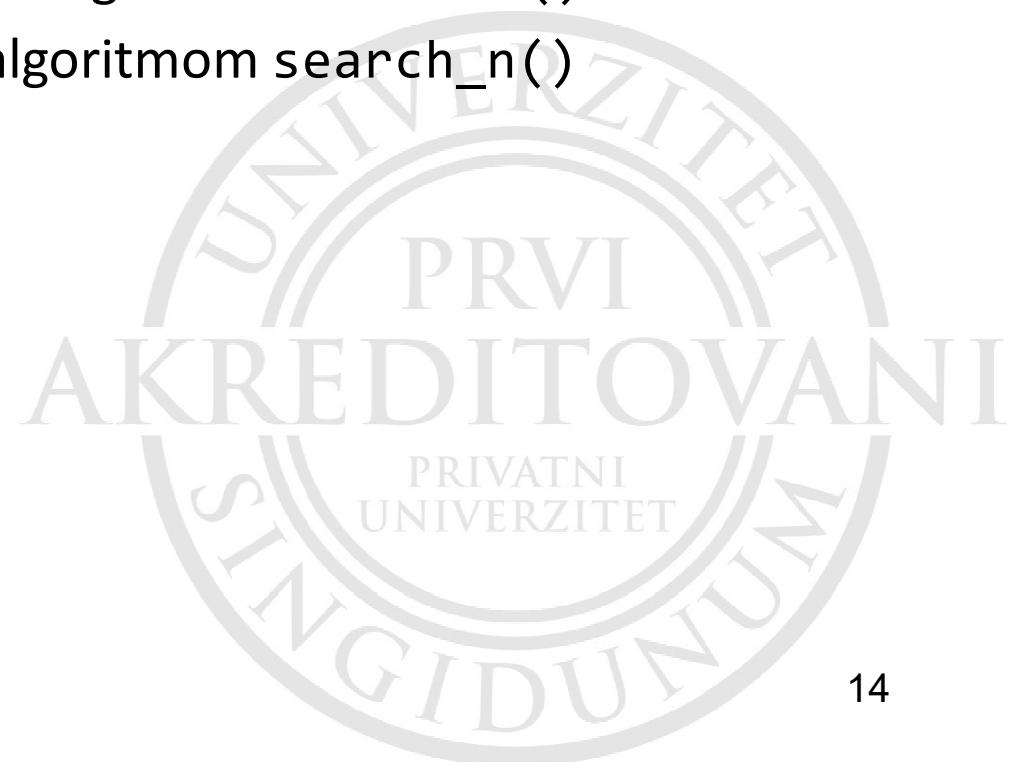
- Pronalaženje u zadanom nizu prve pojave nekog elementa iz drugog niza, za koji važi neki predikat, može se takođe izvršiti algoritmom `find_first_of()`
 - predikat može biti *lambda izraz*, koji se zadaje kao peti argument
 - mogu se porebiti elementi različitog tipa, kad nije definisan operator ==
- Npr. celi brojevi, koji su deljivi nekim od zadanih faktora

```
std::vector<long> brojevi {64L, 46L, -65L, -128L, 121L, 17L, 35L, 91L};  
int faktori[] {7, 11, 13};  
auto iter = std::find_first_of(  
    std::begin(brojevi), std::end(brojevi), // niz  
    std::begin(faktori), std::end(faktori), // traženi  
    [] (long v, long d){return v%d == 0;}); // predikat  
if (iter != std::end(brojevi))  
    std::cout << "Prvi deljivi je " << *iter << std::endl;
```

Prvi deljivi je -65

2.3 Pronalaženje više elemenata drugog niza u nizu objekata

- U zadanim nizu mogu se pronaći *ponavljanja podnizova* (više elemenata drugog niza):
 - prvo *ponavljanje elemenata* algoritmom `adjacent_find()`
 - poslednje *ponavljanje podniza* elemenata algoritmom `find_end()`
 - prvo *ponavljanje podniza* elemenata algoritmom `search()`
 - zadani broj n *ponavljanja podniza* algoritmom `search_n()`



3. Particija niza objekata metodima biblioteke STL

- Particija (podela) elemenata nekog niza je *preuređenje niza*, tako da elementi za koje je neki predikat istinit prethode ostalim elementima niza
- Particija niza objekata može se izvršiti algoritmima
 - `partition()` - vrši podelu niza na dva dela, u skladu s zadanim predikatom, npr. na vrednosti manje od srednje vrednosti niza i ostale
 - `partition_copy()` - vrši podelu elemenata niza na dva dela, ali ne menja originalni niz, već particije kopira u dva posebna niza
 - `partition_point()` - vraća iterator kraja prve particije niza
- Algoritmi se koriste za ubrzavanje operacija nad neuređenim nizovima, kad se ne želi njihovo potpuno sortiranje

4. Algoritmi binarnog pretraživanja biblioteke STL

- Binarno pretraživanje prepostavlja prethodno *sortirane* nizove objekata, jer se pretraživanje zasniva na proveri da li je tekući element manji ili veći od traženog
- Biblioteka šablonu nudi sledeće algoritme binarnog pretraživanja niza objekata
 - `binary_search()` - pretražuje niz zadan pomoću dva iteratora i vraća logičku vrednost *true* ako je element pronađen, inače vraća *false*
 - `lower_bound()` - pretražuje niz zadan pomoću dva iteratora i pronalazi *prvi element* koji nije manji, odnosno veći je ili jednak zadanom elementu
 - `upper_bound()` - pretražuje niz zadan pomoću dva iteratora i pronalazi *prvi element* koji je strogo veći od zadanog
 - `equal_range()` - pronalazi sve elemente jednake zadanom

Algoritam binary_search()

- Algoritam samo ustanavljava da li se traženi element nalazi u zadanim nizu elemenata, npr. u kontejneru tipa dvostruko povezane liste

```
// Sortirana lista celih brojeva u rastućem poretku
std::list<int> lista {11,17,22,36,40,43,48,70,54,61,78,82,89,92,99};
int trazenii {22}; // element koji se traži (nalazi se u listi)
if (std::binary_search(std::begin(lista),
                      std::end(lista),
                      trazenii))
    std::cout << trazenii << " se nalazi u listi" << std::endl;
else
    std::cout << trazenii << " nije u listi" << std::endl;
```

- Algoritam omogućava upotrebu četvrtog argumenta, koji može biti lambda funkcija za poređenje elemenata, npr.

```
[](int a, int b){ return a > b; };
```

lower_bound() i upper_bound()

- Algoritmi pretražuju sortirani niz elemenata i pronalaze prvi element koji je veći ili jednak ili je strogo veći od zadatog
 - elementi su uređeni preko operatora <

```
// Sortirana lista celih brojeva u rastućem poretku
std::list<int> lista {11,17,22,36,40,43,48,70,54,61,78,82,89,92,99};
int trazeni {22};    // element koji se traži (nalazi se u listi)
std::cout << "Donja granica za " << trazeni << " je "
             << *std::lower_bound(std::begin(lista),
                                  std::end(lista), trazeni)
              << std::endl;
std::cout << "Gornja granica za " << trazeni << " je "
             << *std::upper_bound(std::begin(lista),
                                  std::end(lista), trazeni)
              << std::endl;
```

Donja granica za 22 je 22
Gornja granica za 22 je 36

Algoritam equal_range()

- Algoritam pronalazi objekte niza koji su ekvivalentni traženom
 - vraća objekt koji sadrži dva iteratora; prvi iterator pokazuje na element koji nije manji od traženog, a drugi na element koji je veći od traženog
- Efekt je kao da se jednim pozivom pokreću algoritmi lower_bound() i upper_bound(), pa se može napisati

```
// Sortirana lista celih brojeva u rastućem poretku
std::list<int> lista {11,17,22,36,40,43,48,70,54,61,78,82,89,92,99};
int traženi {22};    // element koji se traži (nalazi se u listi)
auto par = std::equal_range(std::begin(lista),
                           std::end(lista), traženi);
std::cout << "Donja granica za " << traženi << " je "
          << *par.first << std::endl;
std::cout << "Gornja granica za " << traženi << " je "
          << *par.second << std::endl;
```

5. Primer

- Implementacija heš tabela



Implementacija heš tabela

- Heš tabele
- Kolizije u heš tabeli
- Heš funkcije



Heš tabele

- Heš tabele predstavljaju strukture koja imaju najbolje teorijske i stvarne performanse u primenama gde su potrebne operacije pronalaženja, umetanja i brisanja elemenata
- Sve operacije se izvršavaju za konstantno srednje vreme $O(1)$, a u najgorem slučaju $O(n)$, kad je potrebna promena veličine heš tabele
- Tabela sadrži elemente koji se smeštaju u polje na poziciju koja se izračunava na osnovu dela sadržaja (ključa)
- Izračunata pozicija se naziva "heš kod", a računa se pomoću pogodno odabrane funkcije, koja treba da obezbedi ravnomernu distribuciju elemenata u polju

Kolizije u heš tabeli

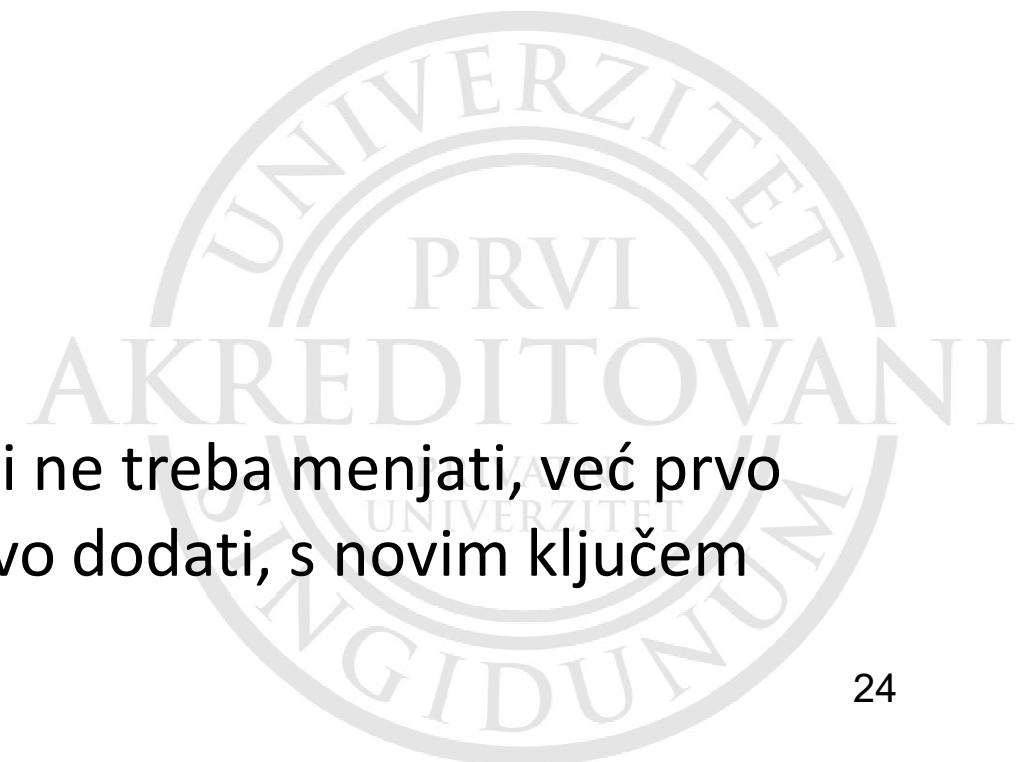
- Kada se za različite vrednosti ključa dobije isti rezultat, događa se "kolizija", koja se razrešava na različite načine, npr. upotrebom prve naredne slobodne pozicije u polju ili smeštanjem elementa u listu kolizija
- Složenost metoda je $O(1+n/m)$ gde je n broj elemenata, a m dimenzija polja (tabele)
- Za $n > m$, kada se tabela popuni, tabelu je potrebno povećati, a elemente prenesti u novu tabelu, uz ponovno računanje heš indeksa (*rehashing*)
- Vremenska složenost ove operacije je $O(n+m)$, ali se ona događa retko, npr. kad se popuni tabela čija se veličina svaki put udvostručava

Heš funkcije

- Osnovni element implementacije heš tabele je pogodna *heš funkcija*, koja uniformno raspoređuje objekte po polju tabele
- Npr. za ključeve tipa `string`, heš funkcija može imati oblik

```
int stringHash(const string& str, int modul) {  
    const int k = 997;  
    int v = 0;  
    for (char c : str) {  
        v = (v * k + c) % modul  
    }  
    return v;  
}
```

- Ključeve elemenata u heš tabeli ne treba menjati, već prvo ukloniti iz tabele, a zatim ponovo dodati, s novim ključem



Perfektne heš funkcije

- Perfektna ili savršena heš funkcija idealno preslikava skup ključeva u potpuno različite elemente, *bez pojave kolizije*
 - to omogućava postizanje konstatnog vremena pristupa elementima heš tabele u praksi
- Minimalna perfektna heš funkcija preslikava n ključeva u n uzastopnih celih brojeva (0.. $n-1$ ili 1.. n)



Praktična primena heš tabela

- Realizacija različitih algoritama visokih performansi, npr.
 - liste simbola prevodilaca
 - traženje anagrama
 - keširanje različitih sadržaja



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Vikipedija www.wikipedia.org
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 15

Algoritmi sortiranja i ažuriranja iz biblioteke šablonu jezika C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

- 1. Uvod**
- 2. Sortiranje nizova metodima biblioteke STL**
- 3. Stapanje (ažuriranje) nizova metodima biblioteke STL**
- 4. Primeri**



1. Uvod

- Sortiranje podataka
- Efikasni algoritmi sortiranja
- Metodi sortiranja u biblioteci STL
- Metodi spajanja u biblioteci STL



Sortiranje podataka

- Sortiranje podataka je veoma čest zadatak programiranja, posebno u obradi transakcija
- Algoritmi sortiranja su neophodni kad je potrebno obezbediti određeni poredak podataka u kontejnerima sekvenci (nizova)
 - asocijativni kontejneri obezbeđuju uređenost elemenata po definiciji
- Metodi sortiranja u biblioteci STL omogućavaju sortiranje bilo kakvih objekata koji se mogu *porediti*, a njihova implementacija je za većinu primena dovoljno *efikasna*
- Sortiranje podataka često prethodi operacijama *ažuriranja*, jer ono može biti efikasnije kad se vrši u istom redosledu u kome su podaci na koje se primenjuju

Efikasni algoritmi sortiranja

- Složenost različitih metoda sortiranja nizova objekata razmatra se u srednjem, najboljem i najgorem slučaju
 - Efikasni su algoritmi čija je srednja vremenska složenost reda $O(n \log n)$
 - Stabilni su algoritmi sortiranja koji čuvaju originalni redosled elemenata koji imaju jednaki ključ, npr. efikasan algoritam *Quick Sort* nije stabilan

Algoritam	Najbolji slučaj	Prosečno	Nagori slučaj	Stabilnost
Quicksort	$n \log n$	$n \log n$	n^2	ne
Merge sort	$n \log n$	$n \log n$	$n \log n$	da
Heapsort	$n \log n$	$n \log n$	$n \log n$	ne
Insertion sort	n	n^2	n^2	da
Selection sort	n^2	n^2	n^2	ne
Bubble sort	n	n^2	n^2	da

Metodi sortiranja u biblioteci STL

- Metodi sortiranja u biblioteci STL pogodni su kao *opšti* metodi sortiranja podataka u STL kontejnerima
 - postoji veliki broj specijalnih metoda sortiranja, koji nisu ugrađeni u standardnu biblioteku jezika C++
- Biblioteka STL u zaglavlju **<algorithms>** sadrži sledeće metode sortiranja:
 - `sort()`
 - `stable_sort()`
 - `partial_sort()`
 - `nth_element()`

Napomena: metod `sort()` obično koristi poboljšanu verziju algoritma *QuickSort*, dok metod `stable_sort` koristi verziju algoritma *MergeSort*



Metodi spajanja u STL biblioteci

- Biblioteka STL u zaglavlju `<algorithms>` sadrži sledeće metode spajanja-ažuriranja nizova objekata:
 - `merge()`
 - `inplace_merge()`



2. Sortiranje nizova metodima biblioteke STL

1. Sortiranje nizova
2. Poredak jednakih elemenata
3. Parcijalno sortiranje
4. N-ti element sortiranog niza
5. Provera sortiranosti nizova



2.1 Sortiranje nizova

- Veliki broj aplikacija zasniva se na *sortiranju* objekata
 - (1) zato što zahtevaju prethodno sortirane podatke ili
 - (2) zbog toga što sortiranje poboljšava njihove performanse
- Šablon sort<Iter>, koji postoji u zaglavlju <algorithms>, sortira nizove elemenata u (podrazumevajućem) *rastućem* poretku, pod uslovom da je definisan operator < za poređenje tipa objekata iz niza
- Uz to, objekti
 - moraju biti međusobno zamjenjivi (*swapable*), pomoću funkcije šablonu swap() definisane u zaglavlju <utility> i
 - moraju imati definisan konstruktor premeštanja (*move constructor*) i operator dodele i premeštanja (*move assignment operator*)

Sortiranje nizova

- Parametar šablonu sort() je tipa iteratora *niza s direktnim pristupom*, tako da je sortiranje elemenata moguće samo u kontejnerima tipa *polja, vektora i dvostrane liste*
 - liste i jednostruko povezane liste imaju funkcije-članove za sortiranje
- Tip elemenata za sortiranje ustanovljava se na osnovu iteratora, koji definišu niz koji treba sortirati, npr.

```
std::vector<int> niz {99, 77, 33, 66, 22, 11, 44, 88};  
std::sort(std::begin(niz), std::end(niz));
```

- Prikaz sortiranog niza može se izvršiti u petlji ili kopiranjem elemenata u izlazni tok algoritmom copy() i iteratora, npr.

```
std::copy(std::begin(niz), std::end(niz),  
         std::ostream_iterator<int> {std::cout, " "});  
// Rezultat: 11 22 33 44 66 77 88 99
```

Sortiranje delova niza i opadajući poredak

- Iteratori omogućavaju sortiranje samo *dela niza*, npr. od drugog do preposlednjeg:

```
std::sort(++std::begin(niz), --std::end(niz));
```

- Sortiranje objekata u *opadajućem* poretku postiže se zadavanjem funkcionskog objekta, koji vrši međusobno poređenje elemenata niza, npr.

```
std::sort(std::begin(niz), std::end(niz),  
         std::greater<>());
```

- funkcija poređenja greater<> vraća rezultat tipa `bool` i ima dva argumenta istog tipa, koji se dobija dereferenciranjem *iteratora* (ili se u njega može implicitno konvertovati)
- funkcija poređenja se može zadati i kao *lambda funkcija*

Primer 1: Sortiranje niza elemenata tipa string u opadajućem poretku

```
#include <iostream>
#include <string>
#include <deque>
#include <algorithm>
#include <iterator>

int main() {
    // Sortiranje niza elemenata tipa string u opadajućem poretku
    std::deque<std::string> reci{ "jedan", "dva", "devet", "devet",
                                "tri", "cetiri", "pet", "sest" };
    std::sort(std::begin(reci), std::end(reci),
              [] (const std::string& s1, const std::string& s2) {
                  return s1.front() > s2.front(); });
    std::copy(std::begin(reci), std::end(reci),
              std::ostream_iterator<std::string> {std::cout, " "});
    std::cout << std::endl;
    return 0;
}
```

tri sest pet jedan dva devet devet cetiri

← obrnuti leksikografski poredak

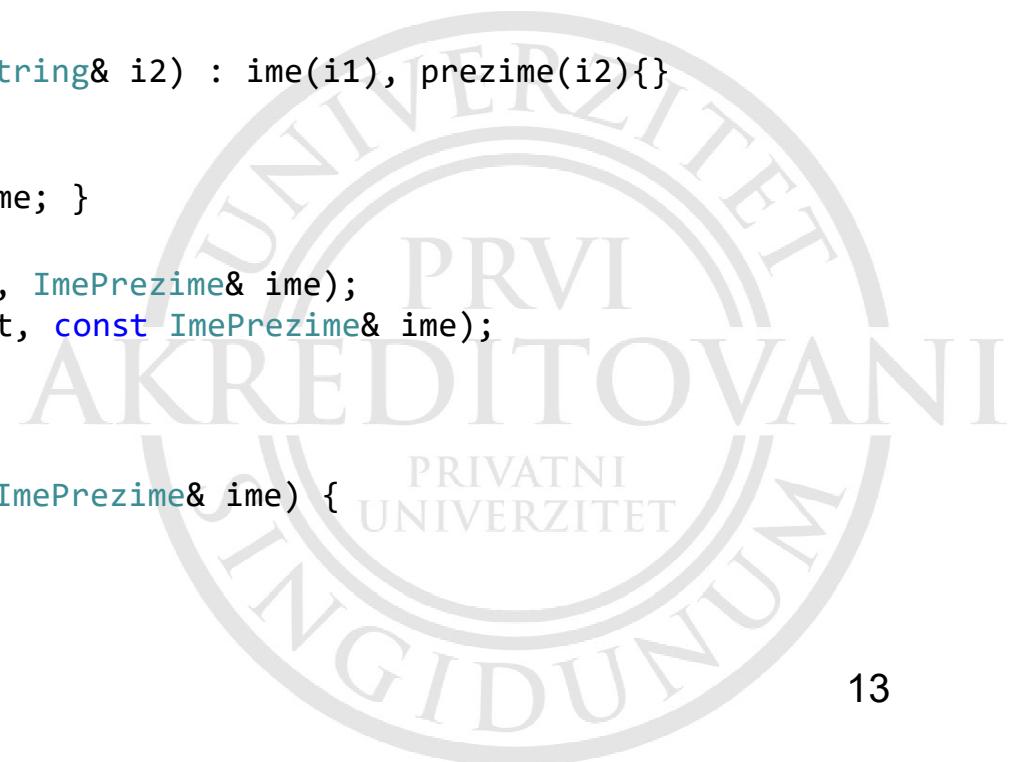
Primer 2: Sortiranje liste imena u rastućem poretku (1/2)

```
#include <iostream>
#include <string>
#include <vector>    // kontejner vector
#include <iterator>   // iteratori stream i back insert
#include <algorithm> // algoritam sort()

class ImePrezime {
private:
    std::string ime{};
    std::string prezime{};
public:
    ImePrezime(const std::string& i1, const std::string& i2) : ime(i1), prezime(i2){}
    ImePrezime()=default;
    std::string get_ime() const {return ime;}
    std::string get_prezime() const { return prezime; }

    friend std::istream& operator>>(std::istream& in, ImePrezime& ime);
    friend std::ostream& operator<<(std::ostream& out, const ImePrezime& ime);
};

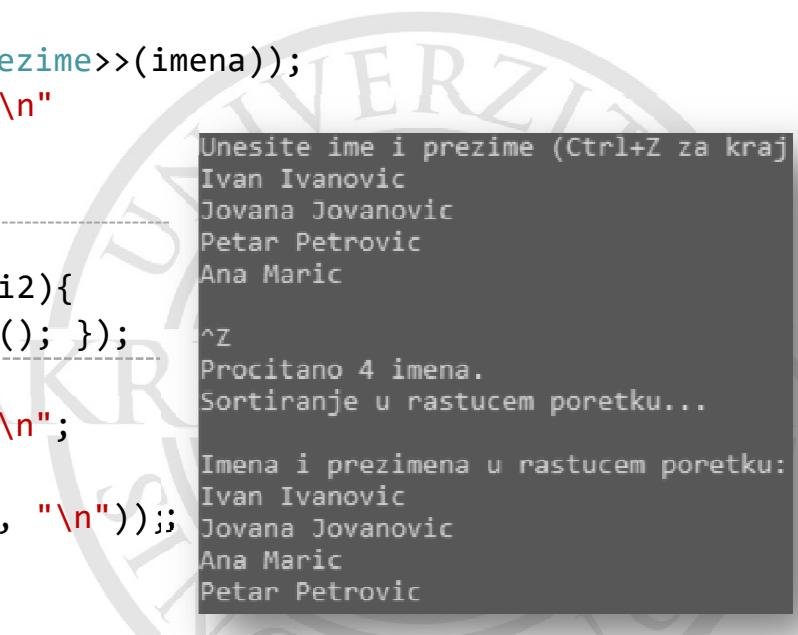
// Ucitavanje objekata ImePrezime
inline std::istream& operator>>(std::istream& in, ImePrezime& ime) {
    return in >> ime.ime >> ime.prezime;
}
```



Primer 2: Sortiranje liste imena u rastućem poretku (2/2)

```
// Prikaz objekata klase ImePrezime
inline std::ostream& operator<<(std::ostream& out, const ImePrezime& ime) {
    return out << ime.ime << " " << ime.prezime;
}

int main() {
    std::vector<ImePrezime> imena;
    std::cout << "Unesite ime i prezime (Ctrl+Z za kraj):";
    std::copy(std::istream_iterator<ImePrezime>(std::cin),
              std::istream_iterator<ImePrezime>(),
              std::back_insert_iterator<std::vector<ImePrezime>>(imena));
    std::cout << "Procitano " << imena.size() << " imena.\n"
          << "Sortiranje u rastucem poretku...\n";
    // Sortiranje po prezimenima u rastucem poretku
    std::sort(std::begin(imena), std::end(imena),
              [] (const ImePrezime& i1, const ImePrezime& i2){
                  return i1.get_prezime() < i2.get_prezime(); });
    // Prikaz sortirane liste imena i prezimena
    std::cout << "\nImena i prezimena u rastucem poretku:\n";
    std::copy(std::begin(imena), std::end(imena),
              std::ostream_iterator<ImePrezime>(std::cout, "\n"));
    return 0;
}
```



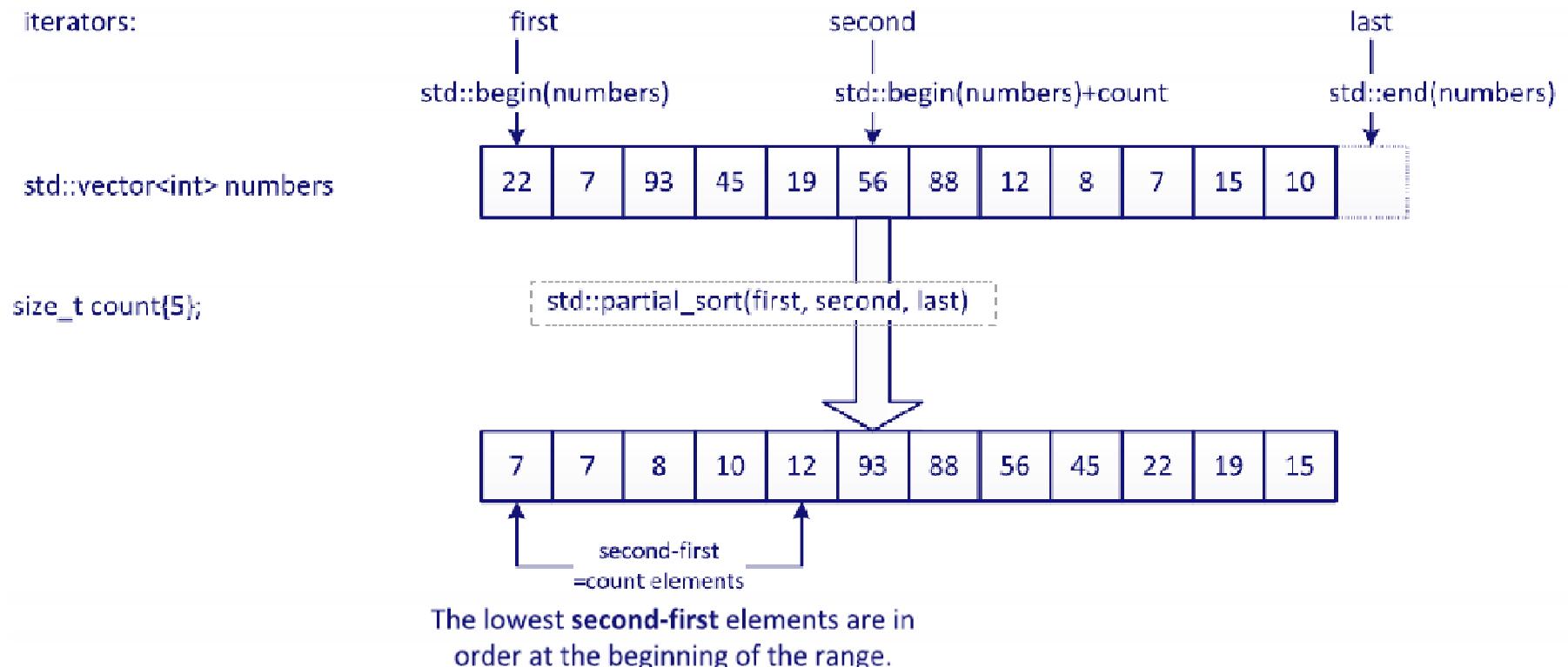
```
Unesite ime i prezime (Ctrl+Z za kraj)
Ivan Ivanovic
Jovana Jovanovic
Petar Petrovic
Ana Maric
^Z
Procitano 4 imena.
Sortiranje u rastucem poretku...
Imena i prezimena u rastucem poretku:
Ivan Ivanovic
Jovana Jovanovic
Ana Maric
Petar Petrovic
```

2.2 Poredak jednakih elemenata

- Standardni algoritam sortiranja *ne garantuje stabilnost*, tako da međusobno jednakim elementima niza ne zadržavaju isti poredak, što može imati neželjene posledice
 - poredak istih elemenata može da nosi neku dodatnu informaciju, koji je u nekim primenama veoma važan. Npr. promena redosleda transakcija istog klijenta može da dovede do prepisivanja podataka
- Algoritam `stable_sort()` garantuje očuvanje originalnog redosleda elemenata koji imaju isti ključ
- Postoje dve verzije algoritma, jedna koja ima dva argumenta za opis niza i druga, koja ima dodatni argument za definisanje načina međusobnog poređenja elemenata

2.3 Parcijalno sortiranje

- Algoritam `partial_sort()` omogućava efikasnije dobijanje sortiranog niza od n najmanjih elemenata iz obimnog niza od N elemenata, $n \ll N$ (originalni redosled jednakih se ne čuva)



Parcijalno sortiranje

- Algoritam parcijalnog sortiranja ima tri argumenta, iteratora s direktnim pristupom
- Primer upotrebe:

```
size_t n {5};    // broj elemenata koji se sortira
std::vector<int>niz{22,7,93,45,19,56,88,12,8,7,15,10};
std::partial_sort(std::begin(niz), std::begin(niz) + n,
                  std::end(niz));
```

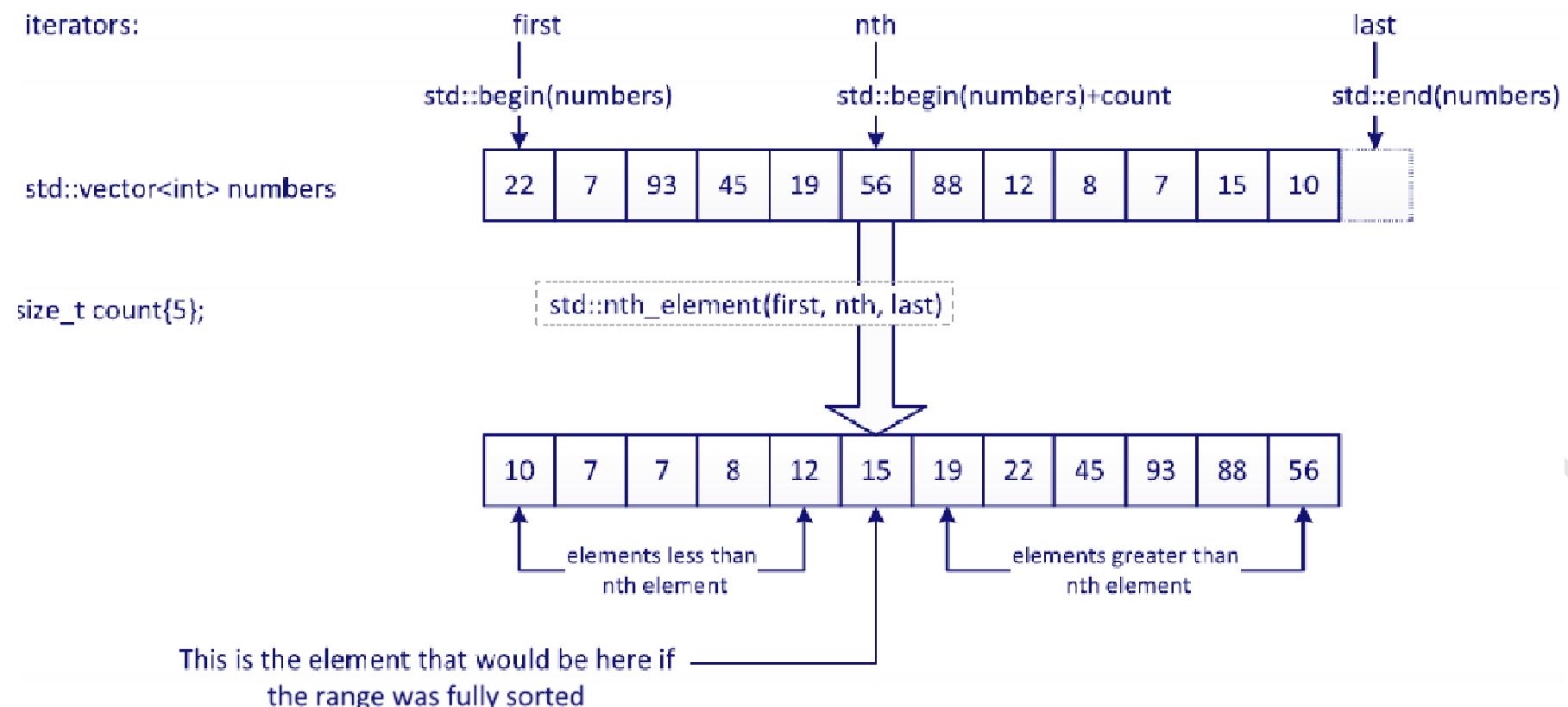
- Dodatni argument se koristi za drugačiji poredak elemenata:

```
std::partial_sort(std::begin(niz), std::begin(niz) + n,
                  std::end(niz),std::greater<>());
```

 - algoritam `partial_sort_copy()` kreira u drugom kontejneru kopiju sortiranog dela niza

2.4 N-ti element sortiranog niza

- Algoritam `nth_element()` primenjuje se na niz definisan prvim i trećim argumentom, dok je drugi argument iterator, koji pokazuje na n -ti element sortiranog niza



2.5 Provera sortiranosti nizova

- Ustanavljanje da li je neki niz već sortiran omogućava izbegavanje nepotrebnih operacija sortiranja
- Funkcija `is_sorted()` vraća *true* ako su elementi niza već sortirani u rastućem poretku. Ako se upotrebi dodatni argument i zada drugačija funkcija poređenja, npr. `greater<>()`, može se proveriti da li je niz sortiran u opadajućem poretku
- Funkcija `is_sorted_until()` vraća iterator koji predstavlja *gornju granicu* niza sortiranog u rastućem redosledu, odnosno prvi element koji je manji od svog prethodnika (ili kraj niza)

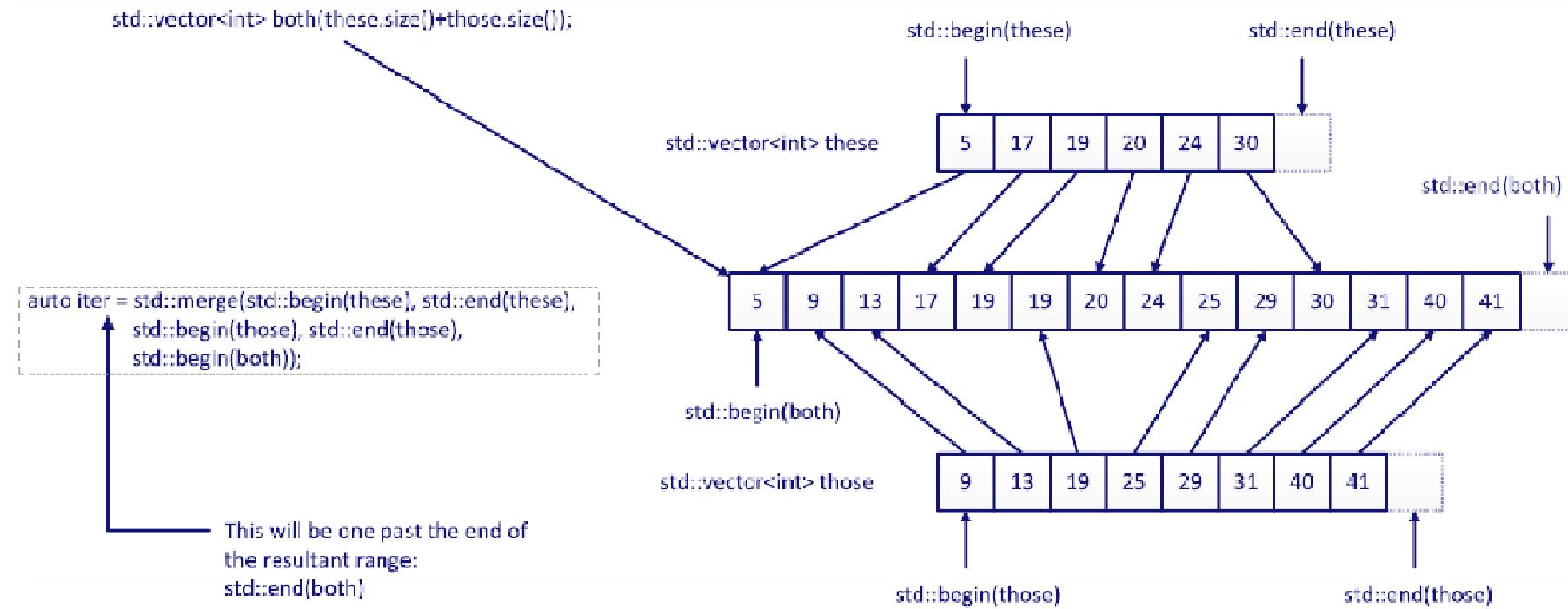
3. Stapanje (ažuriranje) nizova metodima STL biblioteke

1. Stapanje (ažuriranje) nizova
2. Metod spajanja merge()
3. Metod spajanja inplace_merge()



3.1 Stapanje (ažuriranje) nizova

- Operacija stapanja (*merge*) kombinuje elemente dva niza, uređena na isti način, u opadajućem ili rastućem poretku
- Rezultat je novi niz, koji sadrži kopije elemenata dva niza u istom poretku (koristi se operator < za poređenje elemenata)



3.2 Metod spajanja merge()

- Algoritam `merge()` očekuje pet argumenata, koji su iteratori
 - prva četiri argumenta definišu dva niza koji se stapaju - po dva za prvi i drugi niz, a peti argument je iterator koji definiše element kontejnera u koji se smešta prvi element objedinjenog niza
 - ulazni nizovi se ne smeju preklapati (nepredvidivi rezultati)
- Algoritam nema informaciju o kontejneru objedinjenog niza i ne može da kreira njegove elemente, već samo smešta postojeće. To se obezbeđuje kreiranjem objedinjenog niza koji ima broj elemenata jednak zbiru broja elemenata ulaznih nizova ili automatski, pomoću `insert` iteratora
- Algoritam vraća iterator, koji pokazuje na poslednji element objedinjenog niza (peti argument u pozivu funkcije)

Funkcije poređenja Standardne biblioteke

- Može se koristiti i drugačija funkcija poređenja, koja se zadaje kao šesti argument, npr. greater<> iz zaglavlja <functional>
- Ovo zaglavje Standardne biblioteke sadrži *funkcionalne objekte* za aritmetičke, bit i logičke operacije, funkcije negacije, pretraživanja (npr. algoritam Boyer-Moore) i *poređenja*:
 - equal_to realizuje $x == y$
 - not_equal_to realizuje $x != y$
 - greater realizuje $x > y$
 - less realizuje $x < y$
 - greater_equal realizuje $x \geq y$
 - less_equal realizuje $x \leq y$



Primer: Stapanje dva niza u opadajućem poretku

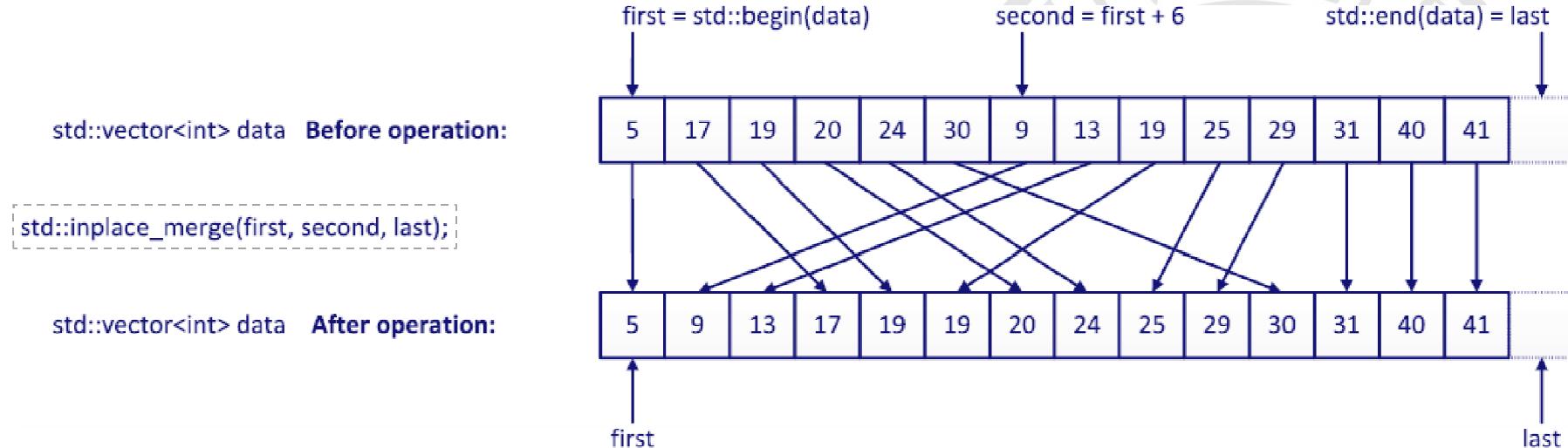
```
#include <iostream>
#include <vector>
#include <iterator>
#include <algorithm>
#include <functional>

int main() {
    std::vector<int> niz1 {2, 15, 4, 11, 6, 7};      // prvi niz
    std::vector<int> niz2 {5, 2, 3, 2, 14, 11, 6}; // drugi niz
    // Sortiranje nizova u opadajućem poretku
    std::stable_sort(std::begin(niz1), std::end(niz1), std::greater<>());
    std::stable_sort(std::begin(niz2), std::end(niz2), std::greater<>());
    // Kreiranje objedinjenog niza dovoljne veličine
    std::vector<int> niz3 (niz1.size() + niz2.size() + 10);
    // Stapanje dva niza u treći, u istom poretku
    auto end_iter = std::merge(std::begin(niz1), std::end(niz1),
                               std::begin(niz2), std::end(niz2),
                               std::begin(niz3), std::greater<>());
    // Prikaz rezultata
    std::copy(std::begin(niz3), end_iter,
              std::ostream_iterator<int>(std::cout, " "));
    return 0;
}
```

15 14 11 11 7 6 6 5 4 3 2 2 2

3.3 Metod spajanja inplace_merge()

- Algoritam `inplace_merge()` objedinjava dva uzastopna sortirana niza elemenata **u istom nizu** u kome se nalaze
 - metod očekuje tri parametra, *prvi*, *drugi* i *poslednji*, koji predstavljaju bidirekcione iteratore. Parametri definišu dva desno otvorena niza elemenata: [*prvi*, *drugi*) i [*drugi*, *poslednji*)
 - rezultat stapanja je niz [*prvi*, *poslednji*)



4. Primeri

1. Pretraživanje i sortiranje: anagrami
2. Obrada kreditnih i debitnih transakcija



4.1 Pretraživanje i sortiranje: lista anagrama

- Anagrami su nizovi reči (izrazi) koji se dobiju permutacijama slova drugih nizova reči, npr. "logaritam" je anagram reči "algoritam"
- Program koji za zadani skup reči vraća skup svih mogućih anagrama tih reči zasniva se na proveri da li je neka reč anagram neke druge reči
- Provera da li je jedna reč anagram druge reči može se efikasno realizovati poređenjem stringova koji se dobiju *sortiranjem* svake od reči, npr. i "logaritam" i "algoritam" nakon sortiranja daju isti string "aagilmort"

Pretraživanje i sortiranje: lista anagrama

- Algoritam pronalaženja anagrama neke reči u zadanom skupu reči poredi svaki string sa ostalim stringovima iz zadanog niza reči. Kad se pronađe string koji je anagram, nije ga potrebno ponovo testirati
- Sortirani string se može koristiti kao ključ *heš table* (kontejner `unordered_map`). Vrednost u tabeli je polje stringova iz ulaznog niza
- Računanje se sastoji od n umetanja u heš tabelu
- Sortiranje ključeva ima složenost $O(n \cdot m \log m)$, a umetanje $O(n \cdot m)$, gde je n broj stringova i m maksimalna dužina stringa. Složenost algoritma pronalaženja anagrama je $O(n^2 \cdot m \log m)$

Lista anagrama (program)

```
#include <iostream>           // tokovi podataka
#include <vector>             // kontejner vector
#include <unordered_map>       // kontejner unordered_map
#include <iterator>            // iteratori
#include <algorithm>          // algoritam sort()
#include <string>              // klasa string
using std::vector;
using std::string;
using std::unordered_map;

// Funkcija pronalazi anagrame u zadanim skupu reci

vector<vector<string>> PronadjiAnagrame(const vector<string>& recnik) {

    std::unordered_map<string, vector<string>> sortiraniStrAnagrami;

    for (const std::string& s : recnik) {
        // Sortira string i koristi ga kao kljuc za dodavanje
        // originalnog stringa kao vrednosti u hes tabelu
        string sortiraniStr(s);
        sort(sortiraniStr.begin(), sortiraniStr.end());
        sortiraniStrAnagrami[sortiraniStr].emplace_back(s);
    }
}
```

Lista anagrama (program)

```
vector<vector<string>> grupeAnagrama;
for (const auto& p : sortiraniStrAnagrami)
    if (p.second.size() >= 2) // ako ima više od jedne reči
        grupeAnagrama.emplace_back(p.second); // pronadjen anagram
return grupeAnagrama;

}

int main() {

    // Skup reci u kojima se traze anagrami
    vector<string> reci { "debitcard", "elvis", "silent", "badcredit",
                          "lives", "freedom", "listen", "levis", "money" };

    vector<vector<string>> anagrami = PronadjAnagrame(reci);

    // Prikaz liste anagrama
    for (auto& grupa : anagrami) {
        for (std::string& rec : grupa)
            std::cout << rec << ' ';
        std::cout << std::endl;
    }
    return 0;
}
```



```
debitcard badcredit
elvis lives levis
silent listen
```

4.2 Obrada kreditnih i debitnih transakcija

- *dodaće se naknadno*



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
5. Horton I., *Beginning C++*, Apress, 2014
6. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
7. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Vikipedija www.wikipedia.org
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Objektno orijentisano programiranje 2

Prof. dr Miodrag
Živković

Tehnički fakultet
2021/2022



Cilj predmeta

- Upoznavanje sa elementima objektno orijentisanog programiranja i programske jezike C++
- Sticanje osnovnih teorijskih znanja o programiranju i osposobljavanje za samostalno rešavanje programske problema kroz praktičan rad na računaru



Ishod predmeta

- Sposobnost snalaženja i rada u različitim razvojnim okruženjima
- Sticanje teorijskih i praktičnih znanja o programiranju i samostalan i grupni rad pri rešavanju programske problema i projekata iz različitih oblasti računarstva



Silabus 2021-2022

1. Uvod u objektno orijentisano programiranje i jezik C++
2. Tipovi i strukture podataka, izrazi i upravljačke naredbe u jeziku C++
3. Funkcije, prenos parametara i dinamička alokacija memorije
4. Klase, objekti, konstruktori i destruktur, nasleđivanje i izvedene klase
5. Polimorfizam i virtuelne funkcije, preklapanje operatora
6. Kolokvijum 1
(e-test/analiza koda)
7. Upravljanje izuzecima
8. Pregled standardne i STL biblioteke
9. Kontejneri
10. Iteratori
11. Tokovi podataka: ulaz/izlaz, rad s fajlovima
12. Kolokvijum 2 (e-test i zadatak)
13. Tehnike efikasnog programiranja u jeziku C++ i uvod u objektno orijentisano modelovanje
14. Algoritmi pretraživanja biblioteke STL
15. Algoritmi sortiranja i ažuriranja biblioteke STL

Ispit

- Metod nastave i savladavanja gradiva
 - predavanja, vežbe, kolokvijumi (e-testovi), ispitni programski zadatak (praktični zadatak u jeziku C++)
- Ispit
 - vrednovanje predispitnih i ispitnih obaveza:

Prisustvo / aktivnost	Kolokvijum 1	Kolokvijum 2	Ispit	UKUPNO
	Teorija i analiza koda	Teorija i praktični zadaci	praktični programski zadatak	
10	30	30	30	100

- prisustvo i aktivnost na predavanjima i vežbama (10)
- kolokvijum 1,2 - imaju teorijski i praktični deo (30+30)
- završni ispit - **ispitni zadatak: program u jeziku C++ (30)**

Nastava

- Predavanja: prof. dr Miodrag Živković
 - ponedeljak, sreda, petak 15⁰⁰-17⁰⁰
- Vežbe: Alekса Ćuk
 - ponedeljak, sreda, petak 17⁰⁰-20⁰⁰



Programski alati

- Microsoft Visual Studio
 - 2010/2012/2013/2015/2017/2019/Community



*Kompajler i linker za C/C++ je cl.exe
(isporučuje se uz Visual Studio)*

- Programski ili tekst editor
 - npr. Notepad++, Visual Studio Code
- Online programska okruženja/prevodioci
 - cpp.sh (C++14), [OnlineGDB](#) (C++17)



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison-Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
5. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
6. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. *Predavanja i materijali s vežbi*
9. Veb izvori



Tema 01 Uvod u objektno orijentisano programiranje i jezik C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

- 1. Uvod**
- 2. Razvoj i standardizacija jezika C i C++**
- 3. Proces razvoja aplikacija u jeziku C++**
- 4. Sintaksa jezika C++**
- 5. Primer programa**



1. Uvod

1. Razvoj programskih jezika
2. Objektno orijentisano programiranje (podsetnik)
3. Razvoj objektno orijentisanih programskih jezika

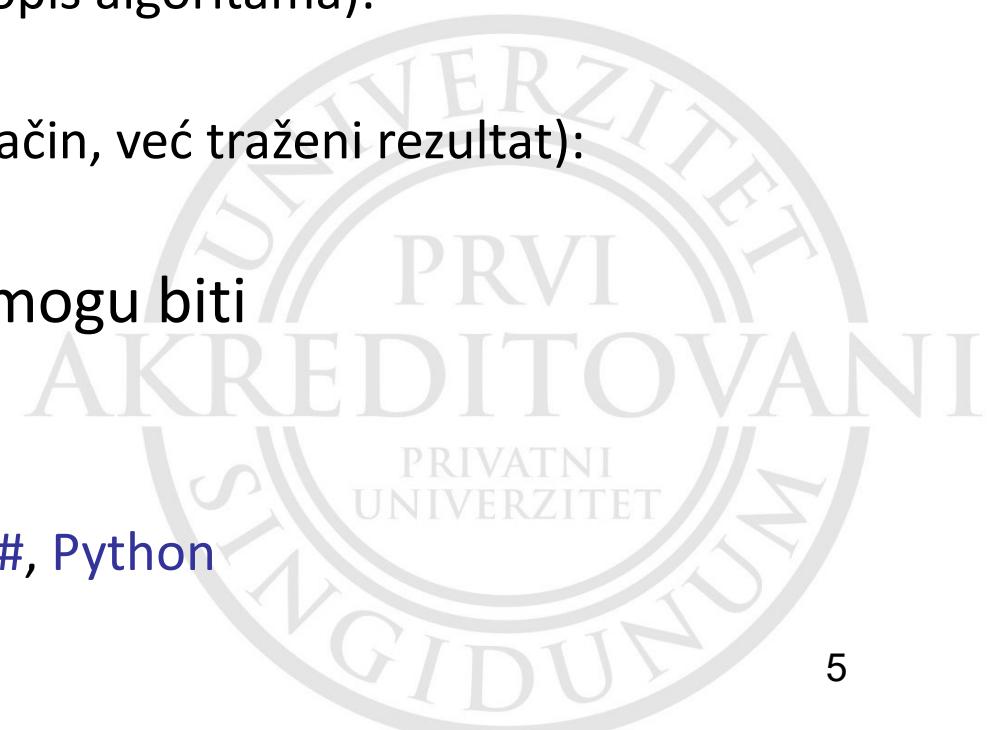


1.1 Razvoj programskih jezika

- Programska jezik je formalni jezik za pisanje programskog koda u obliku niza naredbi koje računar prevodi i izvršava
- Prvi programska jezik s prevodiocem
 - računar Univac i jezik A1, prva poslovna primena računara 1951. godine. Autor jezika je Grays Mary Hooper, kasnije autor jezika COBOL
- Programska jezici za velike centralne računare
 - FORTRAN, COBOL, Algol, Pascal
- Programska jezici za mini i mikroračunare
 - C, BASIC , Visual Basic, Turbo Pascal
- Programska jezici za moderne računare u svetskoj mreži
 - C++, Java, C#, PHP, Javascript, R, Python

Vrste programskih jezika

- Programske jezice niskog nivoa
 - mašinski jezici (binarno kodirane instrukcije)
 - asemblerski jezici (simbolički zapis mašinskih instrukcija i adresa)
- Programske jezice visokog nivoa
 - proceduralni programske jezice (za opis algoritama):
C/C++, Java, C#, Python
 - neproceduralni jezici (ne opisuju način, već traženi rezultat):
Prolog, SQL
- Proceduralni programske jezici mogu biti
 - klasični : **Pascal, C**
 - funkcionalni: **LISP, F#**
 - objektno-orientisani : **C++, Java, C#, Python**



Zašto je potrebno znati jezik C++?

- Jezik C++ je jedan od najpopularnijih programskih jezika za razvoj sistemskog softvera i aplikacija (naredni slajdovi)
- Veliki broj kompanija koristi C++ za razvoj softvera namenjenog za sopstvene potrebe ili tržište (naredni slajdovi)
- Jezik C++ je između mašinski orijentisanih jezika i viših programskih jezika (srednjeg nivoa, *middle-level*)
- C++ je objektno orijentisano proširenje programskog jezika C
- Programi u jeziku C++ su *prenosivi*, jer postoje prevodioci za sve važnije procesore i operativne sisteme
- Jezik C++ je relativno jednostavan jezik (ali ima obimne biblioteke programa)

4.1 Pregled upotrebe programskih jezika opšte namene (IEEE, 2020)

Rank	Language	Type	Score
1	Python▼	🌐💻⚙️	100.0
2	Java▼	🌐📱💻	95.3
3	C▼	📱💻⚙️	94.6
4	C++▼	📱💻⚙️	87.0
5	JavaScript▼	🌐	79.5
6	R▼	💻	78.6
7	Arduino▼	⚙️	73.2
8	Go▼	🌐💻	73.1
9	Swift▼	📱💻	70.5
10	Matlab▼	💻	68.4

Rank	Language	Type	Score
13	SQL▼	💻	64.6
14	PHP▼	🌐	63.8
15	Assembly▼	⚙️	63.7
16	Scala▼	🌐📱💻	63.5
17	HTML▼	🌐	61.4
...	Julia▼	💻	56.0
23	C#▼	🌐📱💻⚙️	48.1
36	Prolog▼	💻	34.6

Web 🌐 Enterprise 💻 Mobile 📱 Embedded ⚙️

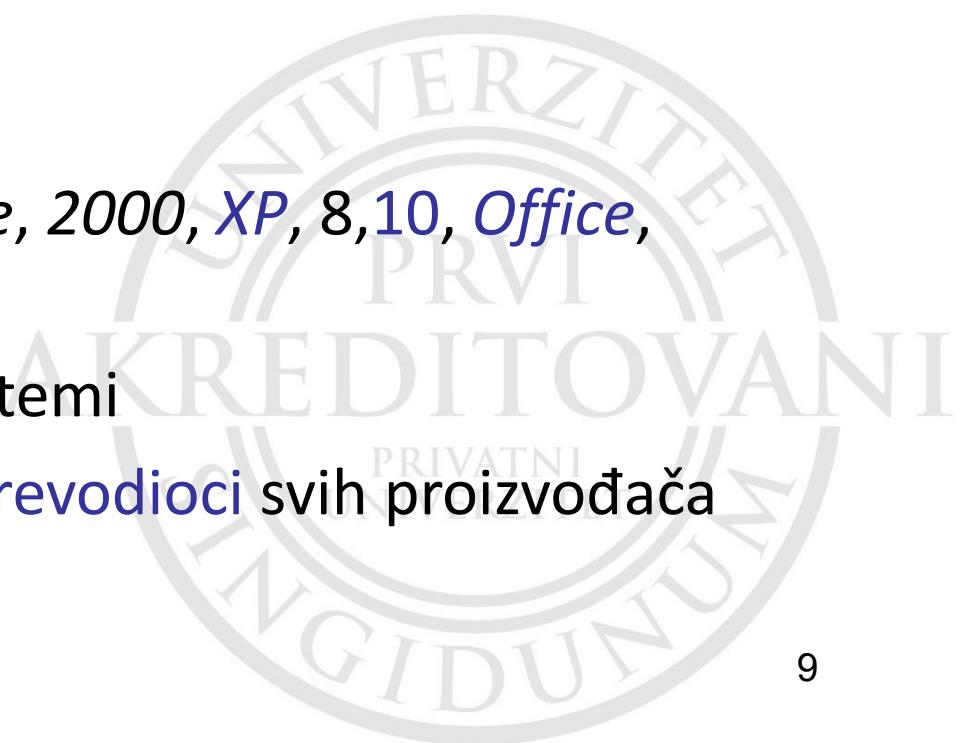
<https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2020>

Ilustracija: Dužina/obim poznatih programa u LOC (*Lines of Code*)

		400.000
		2.500.000
		65.000.000
• Moderni luksuzni automobili		100.000.000
		
		
	X	
		
		

Neko od poznatijih programa implemetiranih u jeziku C++

- Adobe Systems - *Photoshop, ImageReady, Illustrator, Premier*
- Google - *Google file system, Google Chromium, Google Earth, Picasa, Google Desktop Search, MapReduce*
- Mozilla - *Firefox, Thunderbird*
- MySQL - DBMS
- Apple – *OS X, iPod*
- Microsoft - *Windows 95, 98, Me, 2000, XP, 8, 10, Office, Internet Explorer, Visual Studio*
- Symbian OS i brojni ugrađeni sistemi
- Virtuelne mašine i programski prevodioci svih proizvođača



1.2 Objektno orijentisano programiranje (podsetnik)

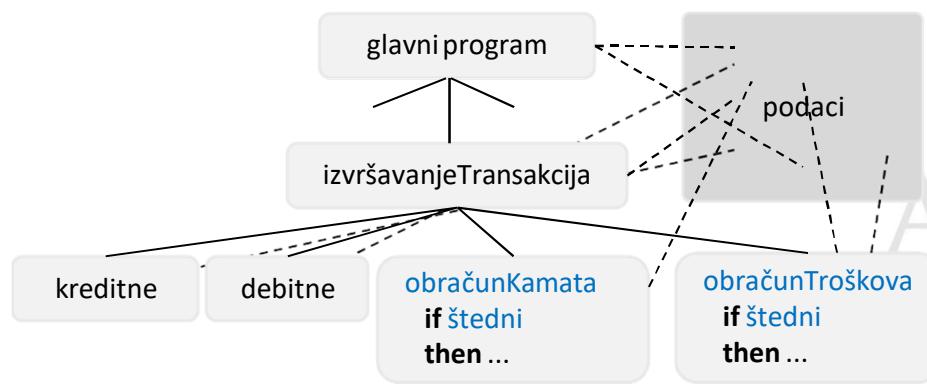
- U **proceduralnom** programiranju, programi se sastoje od funkcija, a podaci se predstavljaju zasebno

Složene interakcije *programskog koda i podataka* iz različitih delova programa takav program čine složenim i teškim za održavanje, pa je za realizaciju složenih softverskih sistema potreban drugačiji pristup
- **Objektno orijentisani pristup** složene softverske sisteme predstavlja kao *skup objekata*, koji se sastoje od *podataka i metoda* kojima se vrše operacije nad objektima

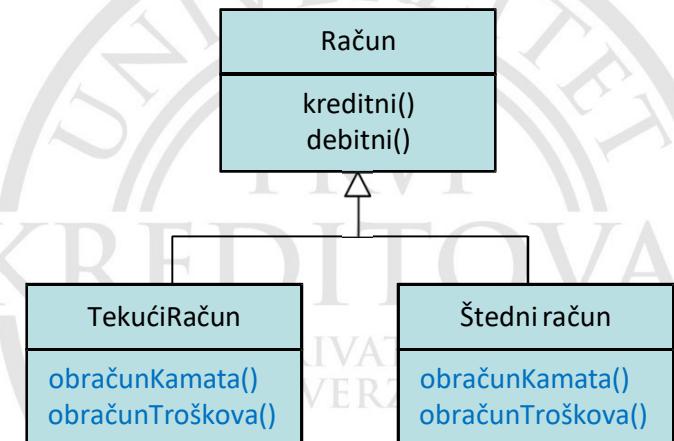
Ovaj pristup organizuje programe na način kako je organizovan stvarni svet, gde su predmeti u *međusobnoj vezi*, kako po atributima, tako i po aktivnostima

Objektno orijentisani razvoj softvera

- Objektno orijentisani pristup rešava mnoge probleme koji su svojstveni proceduralnim programiranju, gde su podaci i operacije su međusobno *razdvojeni*, tako da je neophodno slanje podataka metodima
- Objektno orijentisano programiranje smešta podatke i operacije koje se na njih odnose *zajedno* u objektu



(a) Proceduralna organizacija sistema



(b) Objektna organizacija sistema

Svojstva objektno orijentisanog programiranja i programske jezike

- **Apstrakcija (abstraction)** - mogućnost definisanja novih tipova podataka (klase objekata)
- **Enkapsulacija (encapsulation)** - skrivanje detalja realizacije nekog tipa (klase)
- **Nasleđivanje (inheritance)** - kreiranje novih tipova (klasa) pomoću postojećih, koji će naslediti sve osobine starih tipova i dodati svoje specifičnosti
- **Polimorfizam (polymorphism)** - pojavljivanje tipa (klase) u više oblika, jer se mogu ne samo dodavati svoji elementima, već i menjati nasleđeni

1.3 Razvoj objektno orijentisanih programskih jezika

- Prvi objektno orijentisani jezici
 - Simula, 1967
 - Smalltalk, 1972-1980
- Većina savremenih programskih jezika su objektno orijentisani i mogu biti
 - objektni jezici (*class-based*), koji imaju mogućnost definisanja klasa i nasleđivanje, npr. *C++*, *Java*, *C#* i *Python*
 - objektno zasnovani (*prototype-based*), s ograničenim objektnim svojstvima (kao što je nasleđivanje), koji pretežno koriste postojeće ugrađene objekte, npr. *JavaScript*
- Primeri jezika koji *nisu* objektno orijentisani su *C*, klasični *Pascal* (ne *Object Pascal*) i ostali stariji proceduralni programske jezike

2. Razvoj i standardizacija jezika C i C++

- 1. Razvoj i svojstva jezika C**
- 2. Razvoj jezika C++**



2.1 Razvoj i svojstva jezika C

- Proceduralni programski jezik, čija je prvo bitna namena bila lakši razvoj sistemskih programa
 - jezik je razvio američki naučnik Dennis Ritchie, koji je s kolegom Kenom Thompsonom u *Bell Labs* od 1969. godine razvijao operativni sistem za male računare *Unix* (od *Multics*, operativnog sistema velikih računara)
- Za potrebe ubrzanja razvoja sistemskog softvera razvijeni su novi jezici **B** (Thompson) i kasnije **C** (Ritchie)
 - osnovna inspiracija bio je imperativni proceduralni jezik *Algol* (*Algorithmic Language*)
 - iako se prevodio u mašinski jezik, programi su bili prenosivi, pošto su prevodioci implementirani za mnoštvo različitih platformi
- Jezik **C** je standardizovan 1989. godine (ANSI C/ISO)

Ilustracija: Funkcija faktorijel u jezicima Algol 68, Ada i C

Algol 68

```
PROC factorial = (INT upb n)LONG LONG INT:(  
    LONG LONG INT z := 1;  
    FOR n TO upb n DO z *:= n OD;  
    z  
)
```

Ada

```
function Factorial (N : Positive) return  
    Positive is  
    Result : Positive := N;  
    Counter : Natural := N - 1;  
begin  
    for I in reverse 1..Counter loop  
        Result := Result * I;  
    end loop;  
    return Result;  
end Factorial;
```

C

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

2.2 Razvoj jezika C++

- Danski naučnik Bjarne Stroustrup razvio je jezik '**C s klasama**' **1979.** godine
- Osnovna namera bila je kreiranje jezika koji će imati osobine jezika visokog nivoa kao *Simula* za razvoj složenih sistema i efikasnost jezika niskog nivoa, kao što je bio *BCPL*, prethodnik jezika *B* i *C*
 - radeći na svojoj doktorskoj tezi, imao je problema s lošim stranama oba programska jezika
- Proširio je popularni jezik *C* objektnim svojstvima jezika *Simula*, kao i elementima jezika kao što su *Ada* i *Algol 68*
- Jezik je promenio naziv u **C++** **1983.** godine
- Jezik C++ je standardizovan **1998.** godine (ISO) →

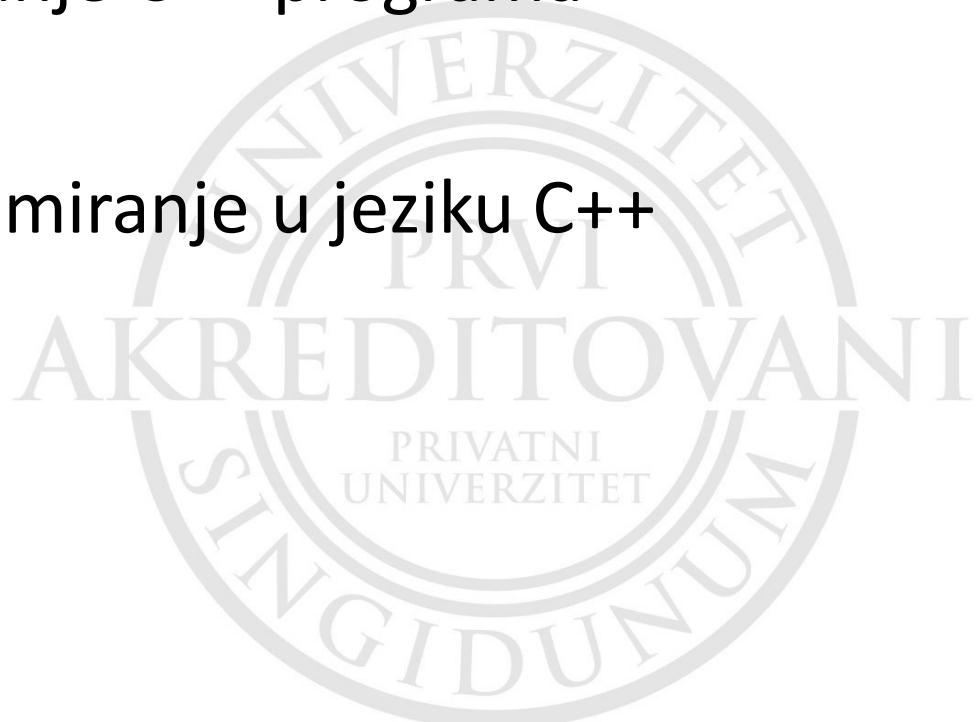
Standardizacija jezika C++

<https://isocpp.org/std>

Godina	ISO standard	Neformalni naziv
1998	ISO/IEC 14882:1998	C++98
2003	ISO/IEC 14882:2003	C++03
2011	ISO/IEC 14882:2011	C++11
2014	ISO/IEC 14882:2014	C++14
2017	ISO/IEC 14882:2017	C++17
2020	ISO/IEC 14882:2020	C++20
2023	<i>nije određen (usvojen plan razvoja)</i>	C++23

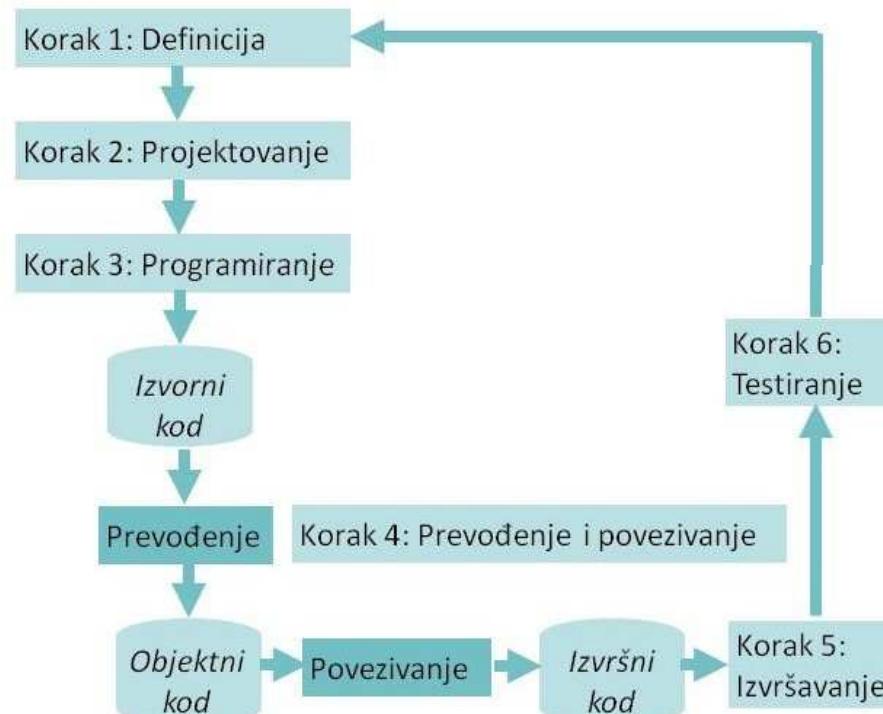
3. Proces razvoja aplikacija u jeziku C++

1. Proces razvoja aplikacija
2. Integrisana razvojna okruženja (IDE)
3. Struktura programa u jeziku C++
4. Unos, prevodenje i izvršavanje C++ programa
5. Pretpocesorske direktive
6. Stilske preporuke za programiranje u jeziku C++



3.1 Proces razvoja aplikacija

- Proces razvoja aplikacija u jeziku C++ obuhvata:
 1. Definisanje problema
 2. Projektovanje rešenja
 3. Programiranje
(pisanje izvornog koda)
 4. Prevođenje u objektni mašinski kod (kompilacija) i povezivanje programa
 5. Izvršavanje programa
 6. Testiranje



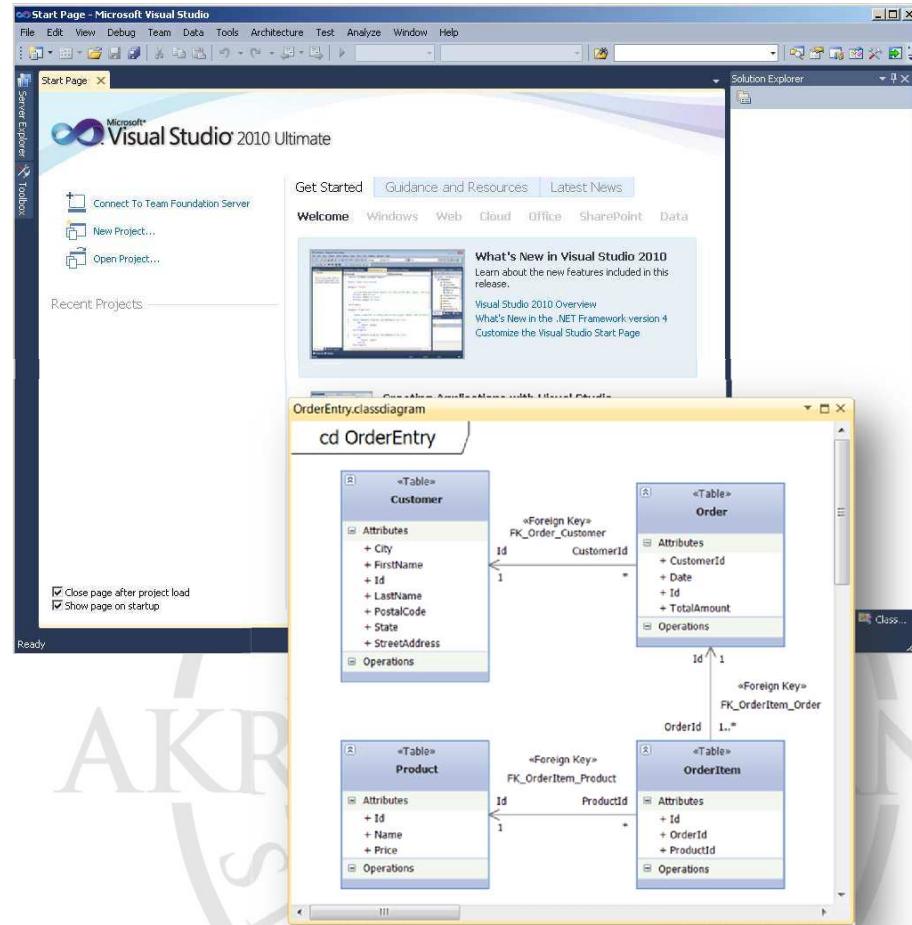
3.2 Integrisana razvojna okruženja (IDE)

- Integrисане programska okruženja povezuju više različitih alata za razvoj softvera pomoću jedinstvenog interfejsa
- Postoji veliki broj integrisanih okruženja za razvoj programa za različite operativne sisteme, npr.
 - Microsoft [Visual Studio](#)/[Visual Studio Community](#)/[Visual Studio Code](#)
 - Eclipse CDT
 - NetBeans C++ IDE
 - C++Builder
 - Code::Blocks
 - CodeLight
 - KDevelop
 - Qt Creator



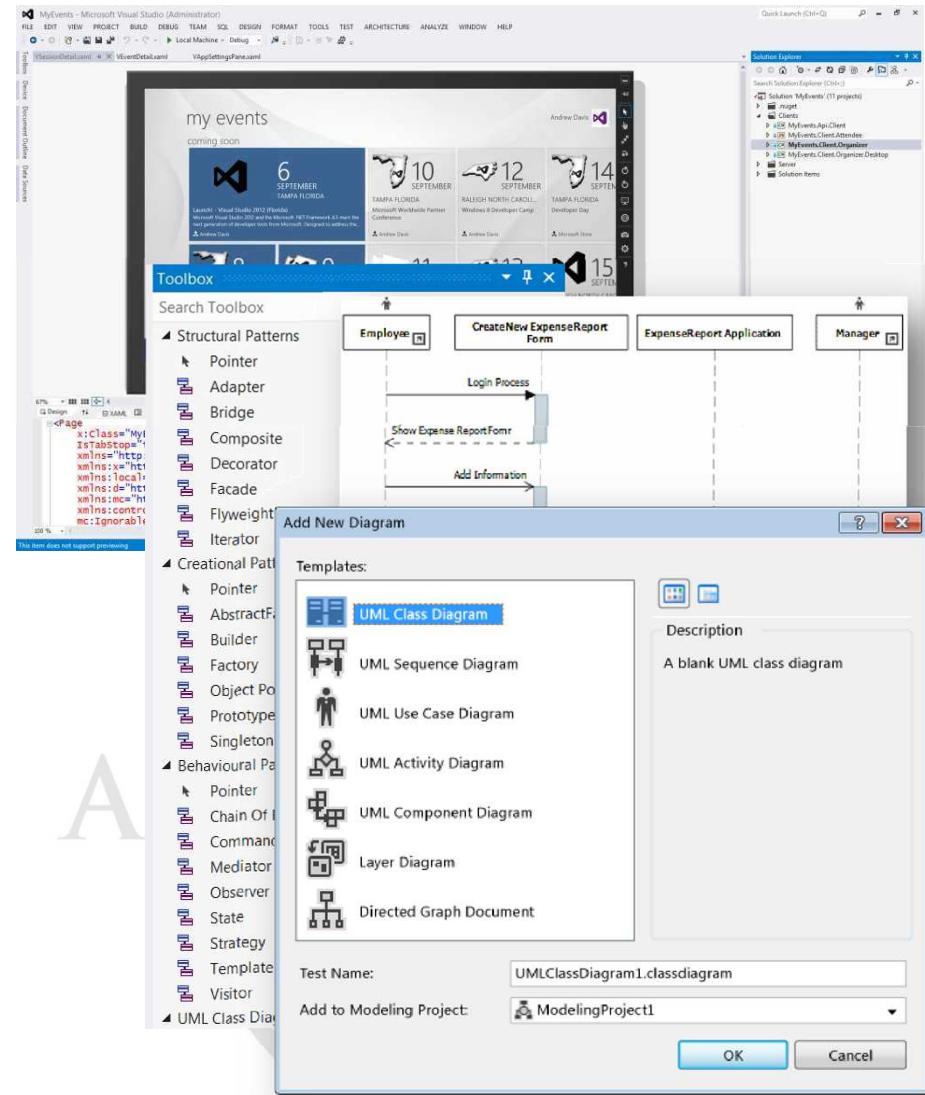
Microsoft Visual Studio 2010

- Integrисани алат, који подрžава све фазе развоја сложених система
- подрžава тимски рад, verzija Ultimate подрžава UML моделовање (6 дјијаграма)
- подрžава више технологија и архитектура ([C++](#), C#, Java, Веб сервиси, SOA)
 - C++03
 - C++11 делimičно

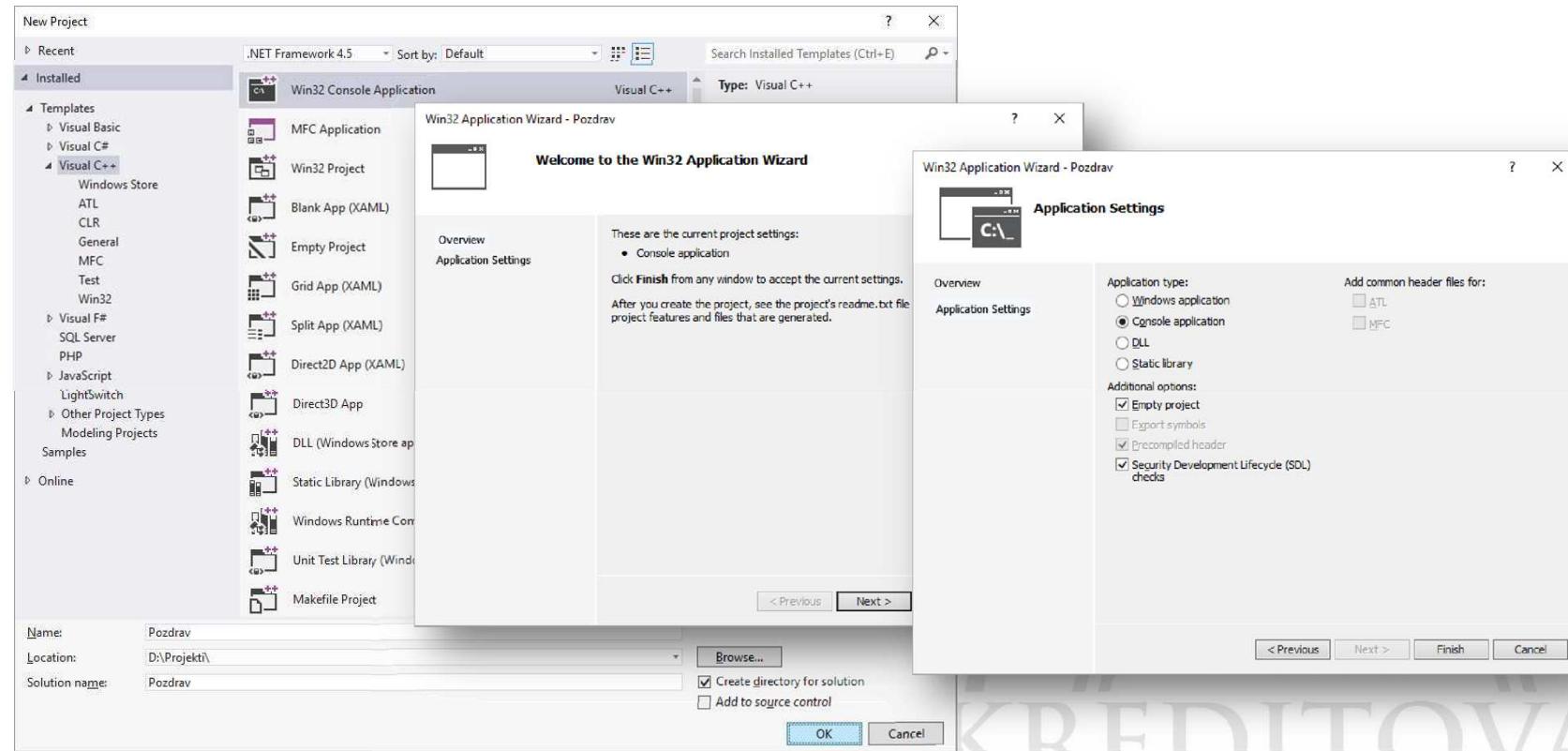


Microsoft Visual Studio 2012-2019

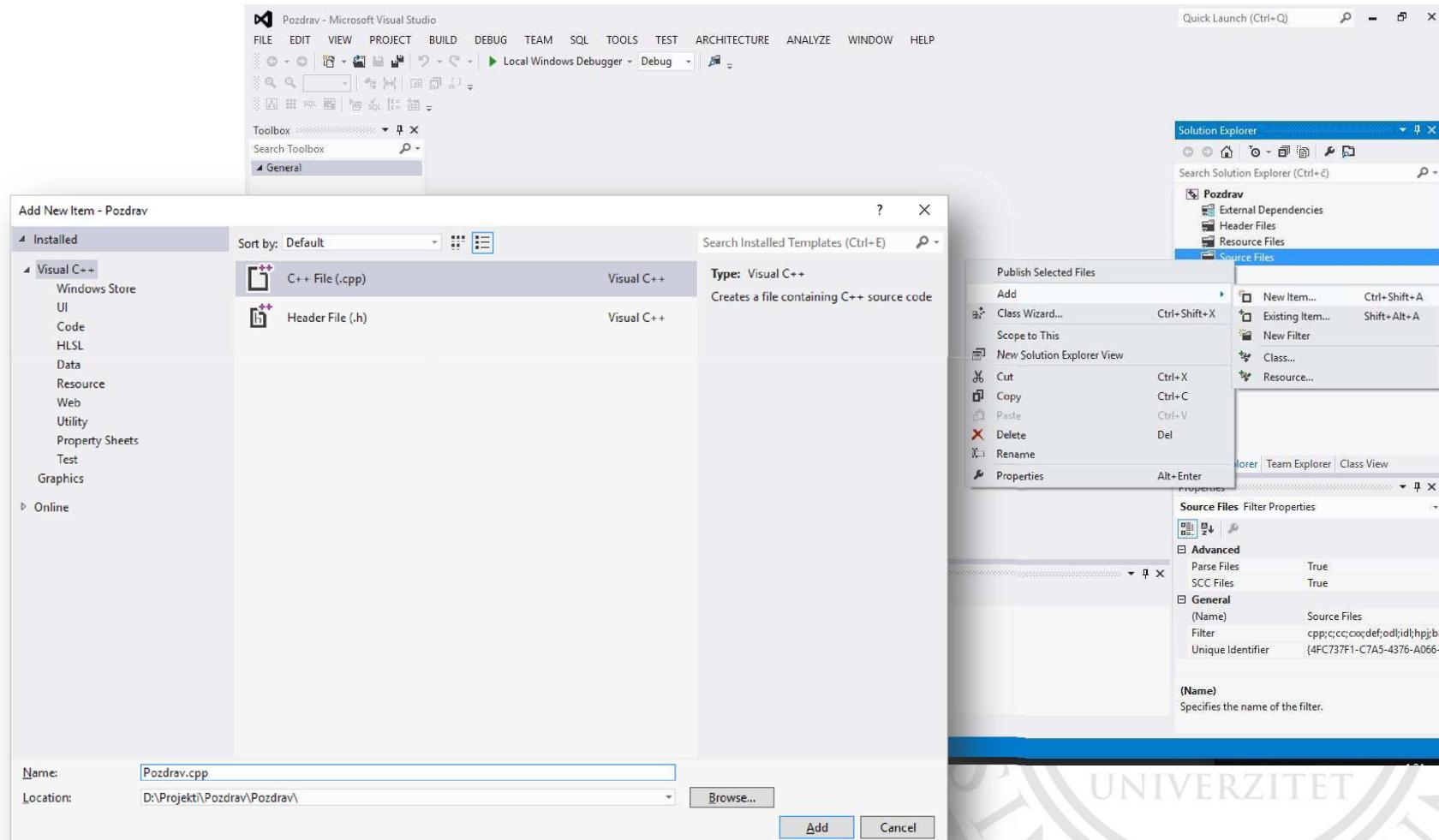
- Integrисани алат, који подрžава све фазе развоја сложених система
- подрžава тимски рад (TFS), verzije Ultimate/Enterprise до 2015 подрžавају UML моделовање (6 дјаграма)
- подрžава више технологија и архитектура ([C++](#), C#, F#, Java, Веб сервиси, SOA)
 - [C++11](#) до [C++17](#)
 - C++20 делimičно



Kreiranje konzolne aplikacije (VS 2012-2013)



Kreiranje C++ programa (VS 2012-2013)



3.3 Struktura programa u jeziku C++

- Jednostavni C++ program:

```
// Prvi program u jeziku C++
#include <iostream> //pretprosesorska direktiva
int main()          //glavna funkcija
{
    std::cout << "Ovo je prvi C++ program!\n";
    return 0;
}
```



Elementi jednostavnog programa: komentar i preprocesorske direktive

- Komentari u jezicima C/C++ mogu biti u jednoj ili više linija

```
// komentar u jednoj liniji
```

```
/* komentar  
u više linija */
```



Elementi jednostavnog programa: komentar i preprocesorske direktive

- Preprocesorska *direktiva*

```
#include <iostream>
```

u kod programa uključuje datoteku **iostream.h** u kojoj su deklaracije funkcija koje vrše ulazno izlazne operacije



Elementi jednostavnog programa: glavni program

- Naredba

```
int main() { ... }
```

definiše programski kôd koji treba da se izvrši kad se pokrene izvršavanje programa

- Velike zgrade u jezicima C/C++ grupišu skupove naredbi u blokove
- Funkcija `main()`
 - ne može biti rekurzivna i ne može se pozivati bilo gde u programu
 - može da ima *argumente* različitih tipova (iz komandne linije)

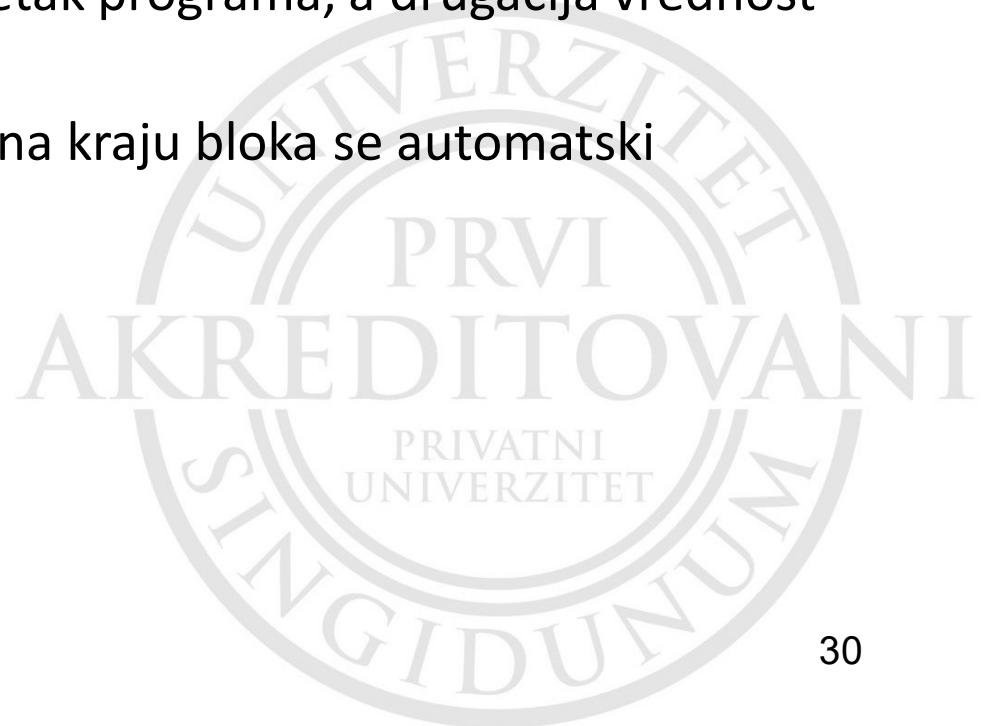
Elementi jednostavnog programa: glavni program

- Naredba

```
return 0
```

završava izvršavanje programa i vraća vrednost 0 procesu koji je pokrenuo program (obično operativni sistem)

- vrednost 0 označava ispravni završetak programa, a drugačija vrednost terminiranje s navedenim kodom
- ukoliko se naredba `return` izostavi, na kraju bloka se automatski izvršava naredba `return 0`



Elementi jednostavnog programa: konzolni ulaz-izlaz

- Naredba

```
std::cout << "Ovo je prvi C++ program!\n";
```

definiše konzolni ispis teksta u standardni izlazni tok std (iostream) . Oznaka << je operator umetanja (*insertion*) podataka u izlazni tok cout (*console out*) - ekran računara

- Ukoliko se na početak programa doda naredba

```
using namespace std;
```

kojom se definiše oblast imena koja će se u programu koristiti, dovoljno je napisati samo cout

- Analogno, standardni ulazni tok je cin, a operator izdvajanja (*extraction*) >> asocira na smer toka podataka

Struktura programa u jeziku C++ (2)

- Jednostavni C++ program s imenikom std:

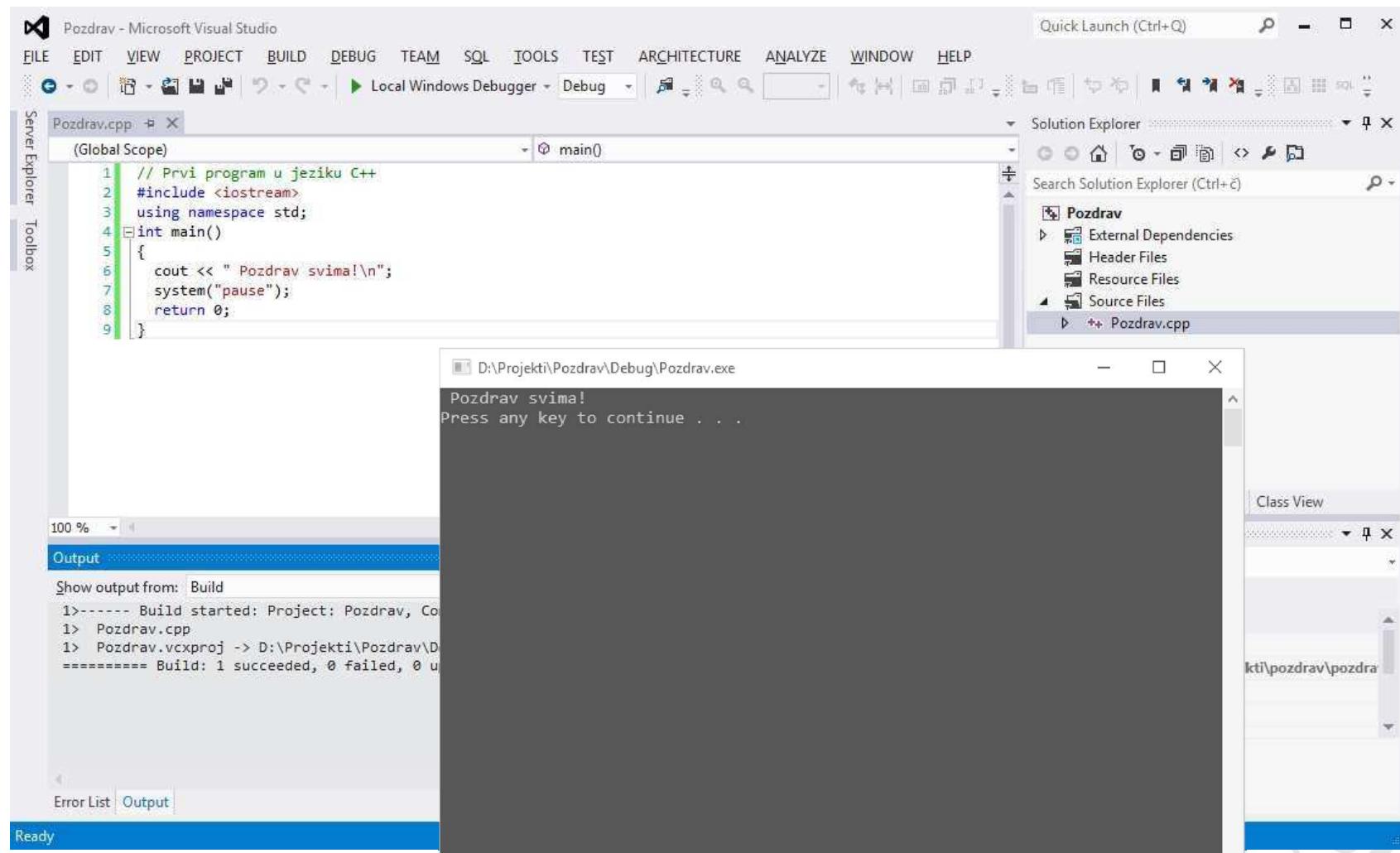
```
// Prvi program u jeziku C++
#include <iostream> //pretprocesorska direktiva
using namespace std; //prostor imena (imenik) std
int main()          //glavna funkcija
{
    cout << "Ovo je prvi C++ program!\n";
    return 0;
}
```



3.4 Unos, prevođenje i izvršavanje C++ programa

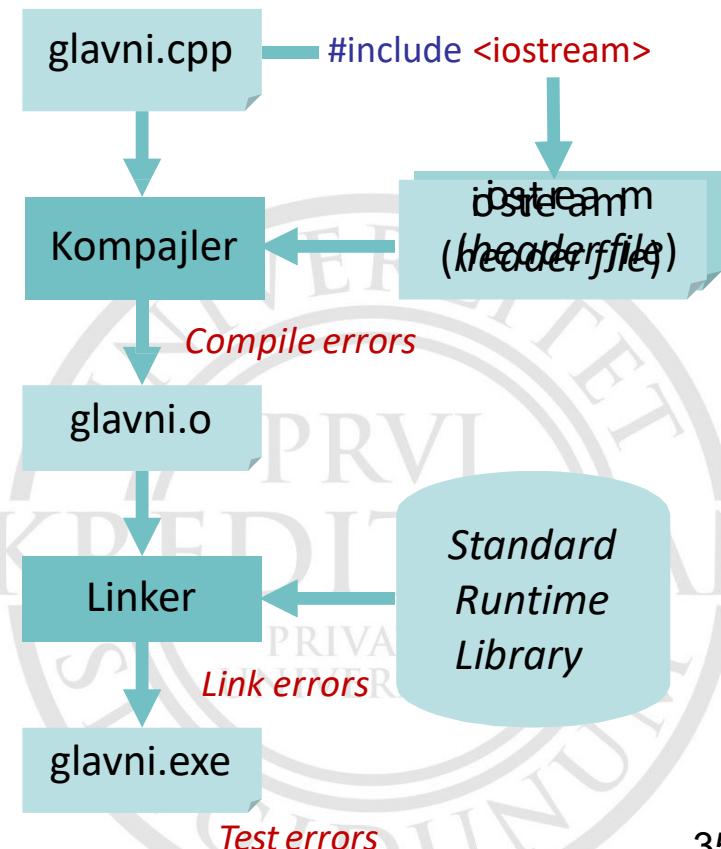
- Unos programskog koda u datoteke sa sufiksom `cpp` pomoću editora ili integrisanog okruženja (IDE)
- Prevođenje, povezivanje i izvršavanje pomoću prevodioca u komandnoj liniji (npr. `cl.exe`) ili pomoću komande integrisanog okruženja (IDE)
- Integrисано окруžење помаже у брјем и лакшем откривању и отклањању програмских грешки (*debugging*)

Unos i izvršavanje jednostavnog programa u jeziku C++



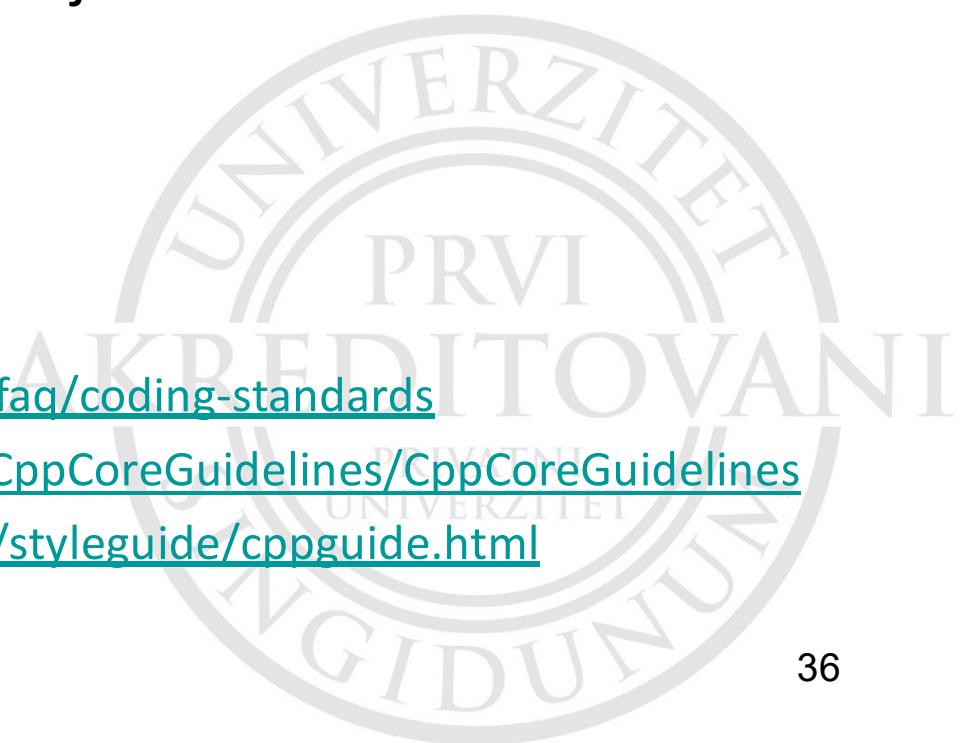
3.5 Pretpocesorske direktive

- Prevodioci programskih jezika C i C++ koriste preprocesor za izmene i dopune izvornog koda *pre prevodenja*
- Opšti oblik direktive je
#direktiva parametri
 - preprocesorske direktive počinju znakom **#** i ne završavaju znakom **;**
 - jedna od najčešćih je direktiva **include**, kojom se u izvorni kod programa uključuje sadržaj drugih datoteka
 - često se koristi i direktiva **define** kojom se definišu konstante i izrazi, npr.
`#define TABLE_SIZE 100`
`#define square(x) x * x`



3.6 Stilske preporuke za programiranje u jeziku C++

- Skup pravila za upotrebu jezika C++, uvek za određenu svrhu i u određenom okruženju
 - ne postoji jedan standard za sve namene i sve korisnike, već se stil prilagođava aplikaciji, kompaniji, području primene i izmenama u jeziku
- Tipični elementi standarda kodiranja
 - strukturiranje koda
 - imenovanje objekata
 - softverski alati
- Standardi i preporuke
 - ISO Standard <https://isocpp.org/wiki/faq/coding-standards>
 - B. Stroustrup <http://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines>
 - Google <https://google.github.io/styleguide/cppguide.html>



Ilustracija: Programski kod na tri načina

Stil 1

```
namespace mojprostor
{
    bool ima_faktor(int x, int y)
    {
        int faktor{ nzd(x, y) };
        if (faktor> 1)
        {
```

Stil 2

```
namespace mojprostor {
    bool ima_faktor(int x, int y)
    {
        int faktor{ nzdf(x, y) };
        if (faktor> 1) {
            return true;
        } else {
```

Stil 3

```
namespace mojprostor {
    bool ima_faktor(int x, int y)
    {
        int faktor{ nzd(x, y) };
        if (faktor> 1)
            return true;
        else
```

funkcija nzd(x,y) pronalazi najveći zajednički delilac dva broja

4. Sintaksa jezika C++

1. Identifikatori
2. Osnovni tipovi i strukture podataka
3. Selekcija i iteracija
4. Funkcije i rekurzija
5. Upravljanje memorijom



4.1 Identifikatori

- Objektima programa dodeljuju se imena, tzv. *identifikatori*, koja treba da zadovolje određene pravila:
 - identifikator mora da bude sastavljen od slova engleske abecede (A..Z i a..z), brojeva 0..9 i donje crte (_)
 - prvi znak mora da bude slovo ili donja crta
 - identifikator ne može da bude ključna reč jezika C++ ili alternativna oznaka nekog operatora
 - posebnu namenu ima *dvostruka donja crta*, pa je treba izbegavati, jer se koriste za nazine implementacija jezika C++ i nazine biblioteka

Rezervisane - ključne reči jezika C++ i alternativni nazivi operatora

asm	do	if	return	typedef
auto	double	inline	short	typeid
bool	dynamic_cast	int	signed	typename
break	else	long	sizeof	union
case	enum	mutable	static	unsigned
catch	explicit	namespace	static_cast	using
char	export	new	struct	virtual
class	extern	operator	switch	void
const	false	private	template	volatile
const_cast	float	protected	this	wchar_t
continue	for	public	throw	while
default	friend	register	true	
delete	goto	reinterpret_cast	try	

and	bitand	compl	not_eq	or_eq	xor_eq
and_eq	bit_or	not	or	xor	

4.2 Osnovni tipovi i strukture podataka

- Ugrađeni tipovi podataka su
 1. celi brojevi (**int**)
 2. realni brojevi (**float, double**)
 3. logičke vrednosti (**bool**)
 4. nizovi znakova (**char, w_char**)
 5. **void** (bez vrednosti)
- Odgovaraju tipovima koje koriste procesori, što omogućava razvoj efikasnog programskog koda
- Jezik C++ omogućava konstruisanje složenijih tipova podataka, kao što su nabranja, strukture i klase

4.3 Selekcija i iteracija

- Selekcija: `if, switch`
- Iteracija: `for, while, do-while`
- Ostale upravljačke naredbe:
`break, continue, goto`



4.4 Funkcije i rekurzija

- Program u jeziku C++ predstavlja skup funkcija
- Funkcije vraćaju vrednost određenog tipa; ako ne vraćaju nikakvu vrednost deklarišu se kao tip **void**
- Svi nazivi funkcija su *globalni*
- Dozvoljena je rekurzija



4.5 Upravljanje memorijom

- Više načina, nema automatizma (skupljanja smeća, *garbage collection*), već je programer odgovoran za upravljanje memorijom (**new/delete**)



5. Primeri programa

- 1.** Implementacija minimalnog programa u različitim programskim jezicima
- 2.** Implementacija programa s dinamičkim poljem u jeziku C++



5.1 Implementacija minimalnog programa u različitim programskim jezicima

- Primer programa koji samo ispisuje na ekran (konzolu) poruku "Pozdrav svima!"
 - implementacija u tri različita savremena programska jezika
 - programi u jeziku C++ prevode se u *mašinski kod* ciljnog računara (efikasniji)
 - programi u jeziku Java i Python se prevode u *međukod (bytekod)*
 - program u jeziku Python je najkraći i najjednostavniji

C++

```
#include <iostream>
using namespace std;
int main() {
    cout << "Pozdrav svima!" << endl;
}
```

Java

```
public class PozdravSvima {
    public static void main(String[] args)
    { System.out.println("Pozdrav
        svima!");
    }
}
```

Python

```
print("Pozdrav svima!")
```

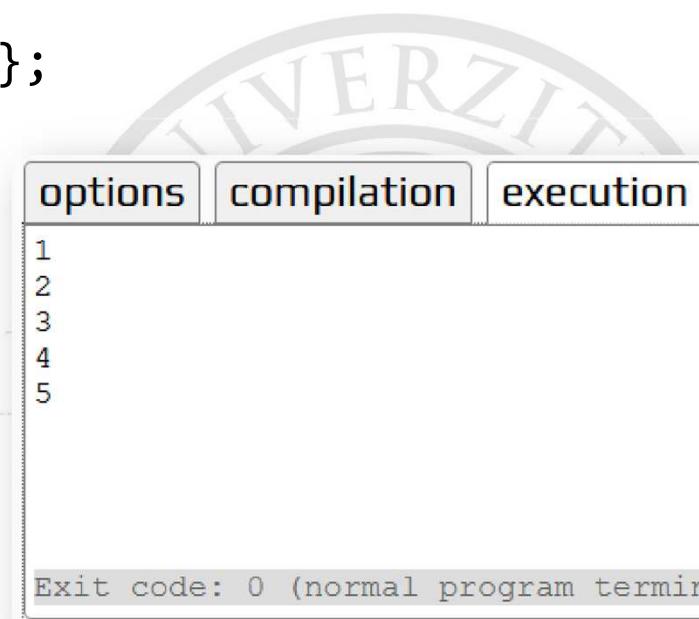
5.2 Implementacija programa s dinamičkim poljem u jeziku C++

```
// Kreiranje, sortiranje i prikaz liste celih brojeva
#include <iostream>
#include <vector>
#include <algorithm>

int main () {

    std::vector<int> v = {2, 1, 5, 3, 4};
    std::sort(v.begin(), v.end());
    for (auto e : v)
        std::cout << e << std::endl;

}
```



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
5. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
6. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. Predavanja i materijali vežbi
9. Veb izvori
 - <http://www.stroustrup.com/>
 - <https://en.wikipedia.org>
 - <https://isocpp.org/>
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 02

Tipovi i strukture podataka, izrazi i upravljačke naredbe u jeziku C++

Prof. dr Miodrag Živković

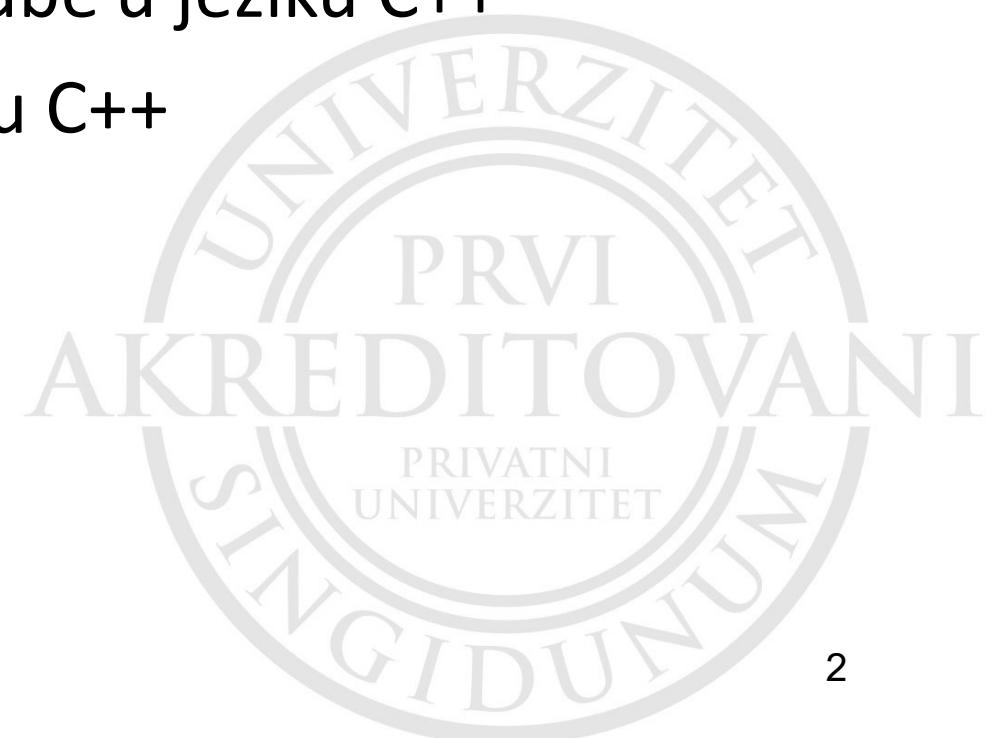
Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Ugrađeni tipovi podataka u jeziku C++
3. Operatori i izrazi u jeziku C++
4. Osnovne upravljačke naredbe u jeziku C++
5. Strukture podataka u jeziku C++
6. Primeri programa



1. Uvod

1. Elementi proceduralnih programskega jezika
2. Identifikatori i specijalni znakovi u jeziku C++
3. Promenljive u jeziku C++



1.1 Elementi proceduralnih programske jezika

- Osnovni elementi proceduralnih programskih jezika su programske **naredbe**, koje načelno odgovaraju rečenicama prirodnog jezika
- Naredbe u jeziku C++ završavaju znakom ";", npr.
 - deklarativne naredbe
`int x;`
 - naredba dodele vrednosti
`x = 10;`
 - ulazno-izlazne naredbe
`cout << x;`
 - ostale naredbe: upravljačke, definicije struktura, ...

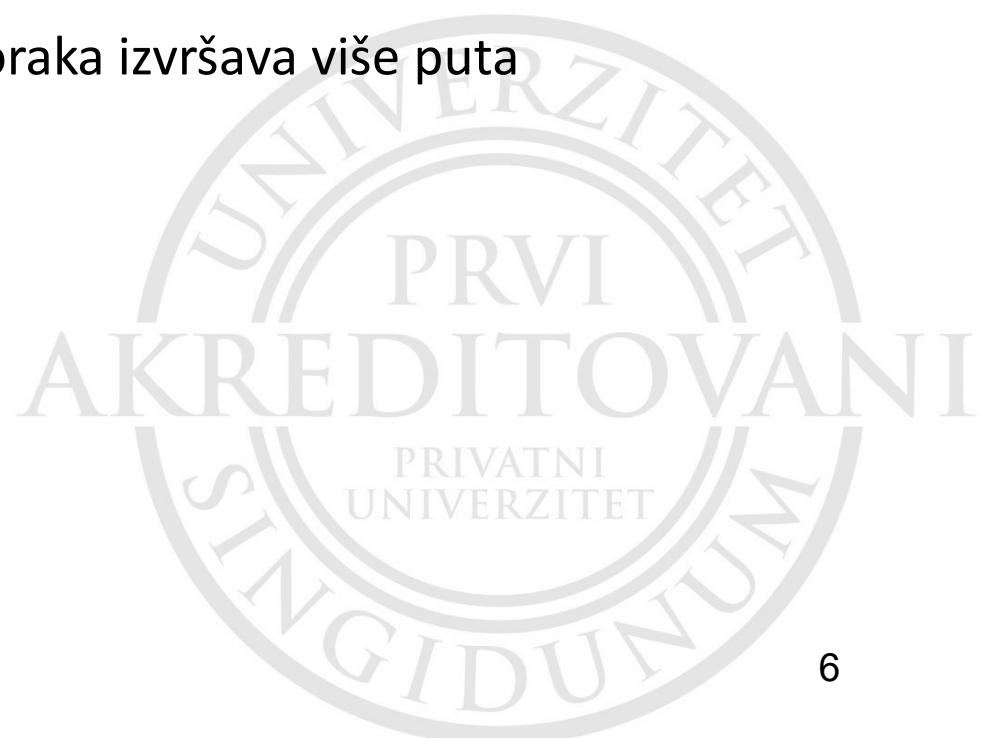


Elementi proceduralnih programskega jezika

- Elementi naredbe za dodelu vrednosti su **promenljive** i **izrazi**
- **Izrazi** se sastoje od *promenljivih, konstanti, operatora* i poziva *funkcija*
 - **funkcije** su imenovane grupe naredbi, koje se mogu višestruko upotrebiti i program čine jasnijim i lakšim za održavanje
 - glavni program u jeziku C++ se definiše kao funkcija **main()**
- Za predstavljanje podataka koriste se
 - **osnovni tipovi** podataka (**int, float, double, bool, char**) koji odgovaraju pojedinačnim vrednostima
 - **strukture** podataka, gde je više vrednosti zapamćeno pod jednim nazivom. Zapamćene vrednosti mogu biti istog tipa (npr. polja) ili različitih tipova (korisnički definisani tipovi)

Elementi proceduralnih programskega jezika

- Za realizacijo bilo kog algoritma u proceduralnom programskom jeziku dovoljna su tri načina upravljanja izvršavanjem pojedinih koraka algoritma
 - sekvenčno izvršavanje, jedan korak za drugim (podrazumeva se)
 - uslovno izvršavanje, gde naredni korak zavisi od određenih uslova
 - ponavljanje, gde se određeni niz koraka izvršava više puta



1.2 Identifikatori i specijalni znaci u jeziku C++

- Elementi jezika imaju nazine (identifikatore), koji se formiraju od slova engleske abecede (A..Z i a..z), cifara 0..9 i znaka "_", s tim da prvi znak mora biti slovo ili donja crta
 - najveća dužina identifikatora je ≥ 1024 karaktera (Microsoft C++ 2048)
- Specijalni znaci (*escape sequences*) navode se posebnim oznakama, koje počinju obrnutom kosom crtom, npr.

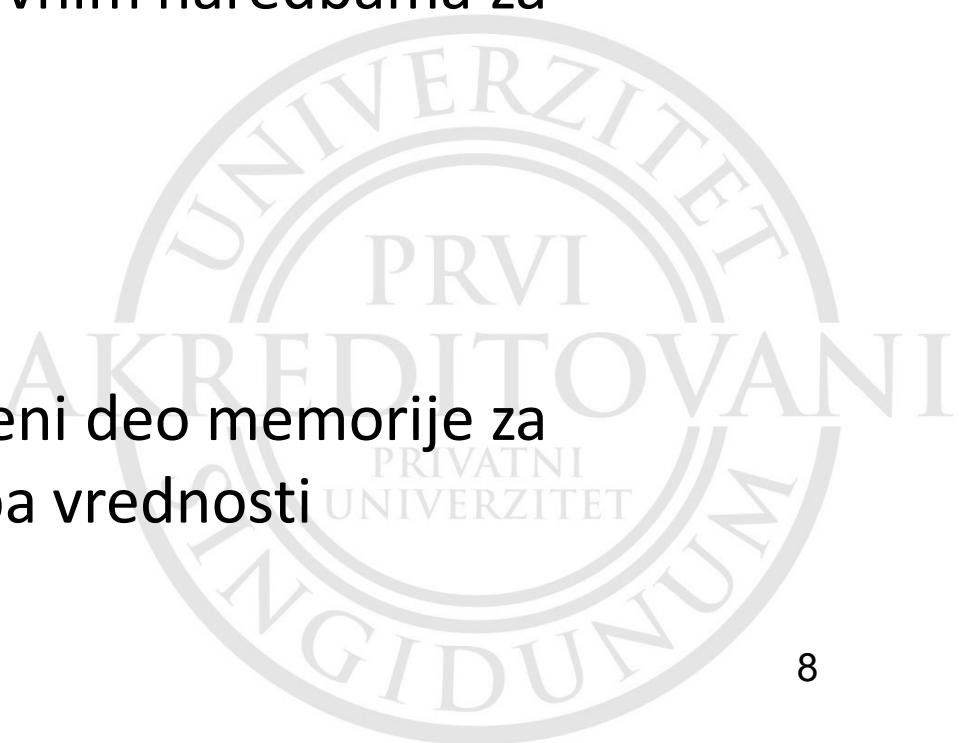
Specijalni znak	Naziv	Opis
\n	newline	Pomera kurSOR u sledeći red
\t	tab	Pomera kurSOR na sledeći tabulator
\a	alarm	Računar se oglašava
\b	backspace	Vraća kurSOR za jednu poziciju unazad
\r	carriage return	Pomera kurSOR na početak tekućeg reda
\\\	backslash	Prikazuje obrnutu kosu crtu
\'	single quote	Prikazuje apostrof
\\"	double quote	Prikazuje navodnik
\?	question mark	Prikazuje znak pitanja

1.3 Promenljive u jeziku C++

- Naredbe u jeziku C++ kreiraju, upotrebljavaju i brišu različite programske objekte. Svaki objekt zauzima odgovarajući prostor u radnoj memoriji. Većina objekata se koristi u obliku imenovanih objekata - **promenljivih**
- Promenljive se kreiraju deklarativnim naredbama za definisanje promenljivih, npr.

```
int x
float y
char z
```

kojima se samo rezerviše određeni deo memorije za promenljivu zadanoj naziva i tipa vrednosti



Promenljive u jeziku C++

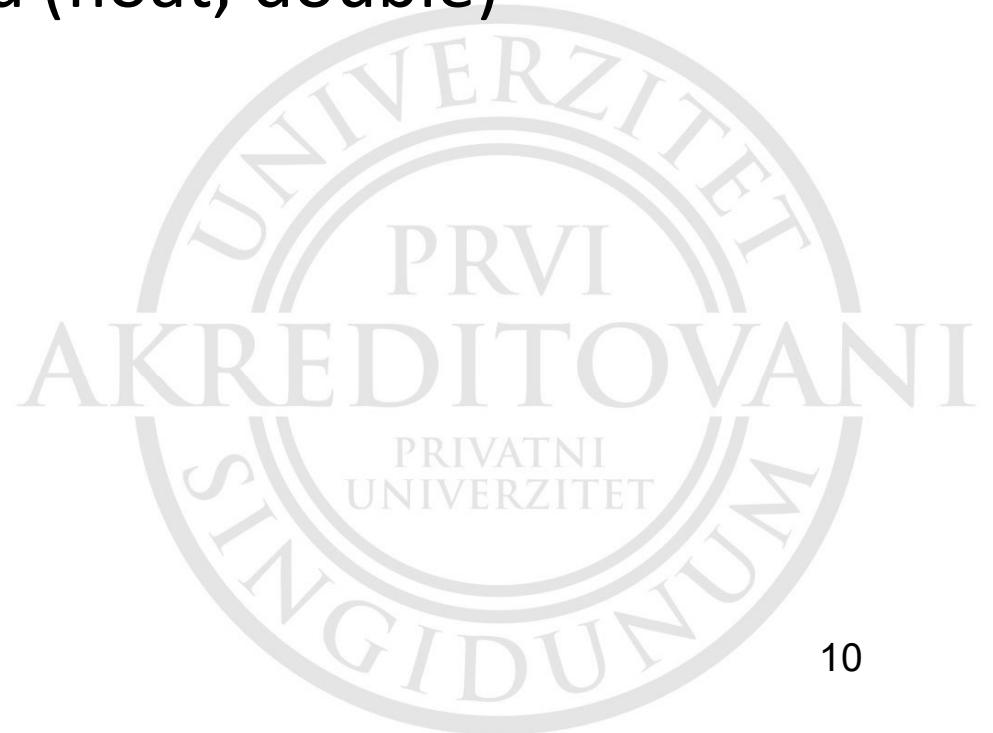
- U jeziku C++ ne vrši se automatska *inicijalizacija* deklarisane promenljive, već je to odgovornost programera
- Razlog su performanse, mada rad s neinicijalizovanim promenljivima daje nepredvidljive rezultate, npr.

```
int z;  
cout << z << endl;
```

- Prema konvenciji, naziv promenljive počinje malim slovom, a ako se sastoji od kombinacije više reči, one se odvajaju donjom crtom ili velikim početnim slovima (*CamelCase*)
- Imena treba da budu *opisna*, tako da treba izbegavati trivijalna (*i, x*) i suviše opšta imena (*brojac, podatak*)

2. Ugrađeni tipovi podataka u jeziku C++

1. Promenljive i konstante
2. Celobrojni podaci (int)
3. Znakovni podaci (char)
4. Brojevi u pokretnom zarezu (float, double)
5. Logički tip (bool)



Osnovni tipovi podataka i proširenja

- Jezik C++ podržava sedam osnovnih tipova podataka:
 - `char` - niz znakova kodirnih pomoću 8 bita
 - `wchar_t` - niz znakova kodiranih s više od 8 bita (*wide character*)
 - `int` - celi broj
 - `float` - decimalni broj
 - `double` - decimalni broj dvostrukе preciznosti
 - `bool` - logička vrednost
 - `void` - bez vrednosti
- Predviđena je mogućnost proširenja osnovnih tipova pomoću *modifikatora* (`signed`, `unsigned`, `long` i `short`), koji se navode ispred naziva tipa

Dozvoljene kombinacije osnovnih tipova podataka i modifikatora

Tip	Veličina u bajtovima	Opseg definisan ANSI/ISO standardom
bool	1	true ili false
char	1	-128 do 127
unsigned char	1	0 do 255
signed char	1	-128 do 127
int	4	-2 147 483 648 do 2 147 483 647
unsigned int	4	0 do 4 294 967 295
signed int	4	isto kao int
short int	2	-32 768 do 32 767
unsigned short	2	0 do 65 535
int		
signed short int	2	isto kao short int
long int	4	-2 147 483 648 do 2 147 483 647
signed long int	4	isto kao long int
unsigned long	4	0 do 4 294 967 295
int		
float	4	$\pm 3,4 \times 10^{-38}$ sa tačnošću od približno 7 cifara
double	8	$\pm 1,7 \times 10^{-308}$ sa tačnošću od približno 15 cifara
long double	8	isto kao double

Veličina u bajtovima zavisi od arhitekture računara i prevodioca.
Standard definiše minimalnu veličinu

Deklaracija i definisanje promenljive

- Deklaracija svake promenljive u programu je **obavezna**, pa je jezik C++ *strogo tipiziran* jezik
- Deklaracija promenljive u jeziku C++ može se navesti *bilo gde* u programu pre prve upotrebe promenljive
- **Deklarisanje** promenljive vrši se navođenjem tipa i naziva promenljive, npr.

```
int tezina; // deklaracija celobrojne promenljive
```

(deklarisanje ne podrazumeva inicijalizaciju)

- **Definisanje** promenljive podrazumeva i *inicijalizaciju*, npr.

```
int tezina = 70;    // definisanje promenljive
int tezina = {70}; // definisanje promenljive u C++11
```

2.1 Promenljive i konstante

- **Promenljive** su podaci određenog tipa čija se vrednost može menjati u toku izvršavanja programa
- **Konstante** (literali) su vrednosti određenog tipa koje se ne menjaju u toku izvršavanja programa, npr.
 - vrednosti tipa **char**: 'A', 'z', '8', '*'
 - vrednosti tipa **int**: -77, 65, 0x9FE
 - vrednosti tipa **unsigned int**: 10U, 64000U
 - vrednosti tipa **long**: -77L, 65L, 12345L
 - vrednosti tipa **unsigned long**: 12345UL
 - vrednosti tipa **float**: 3.14f
 - vrednosti tipa **double**: 3.14
 - vrednosti tipa **bool**: true, false



2.2 Celobrojni podaci (int)

- Celobrojni tip podataka za predstavljanje pozitivnih i negativnih celih brojeva koristi najmanje 2, a tipično 4 bajta
 - `short int` 2 bajta
- Većina procesora podržava celobrojnu aritmetiku u punom komplementu, gde je 16-bitni broj u rasponu -32768..32767
- Standardni celobrojni tip je označen (*signed*), a modifikator *unsigned* treba koristiti samo za predstavljanje pozitivnih vrednosti (prevodilac to ne kontroliše)
- Celobrojne konstante su standardno tipa `int`, pa se tip može izostaviti, npr.

```
brojSoba = 300; // promenljiva brojSoba je tipa int
```

Tip celobrojnog literalna

- Tip celobrojnog literalna može se *definisati* i pomoću tipa dodeljene vrednosti, npr.

```
brojSpratova = 32L;  
brojSoba = 300L;  
brojApartmana = 10u;
```

- Kada tip literalna nije zadat, prevodilac jezika C++ koristi najmanji celobrojni tip koji može da predstavi vrednost, najčešće je to tip **int**
- Konstante se mogu zadati i *heksadecimalno* i *oktalno*, npr.

```
int cetrnaestPatuljaka = 0x0E; // broj 14 (decimalno)  
int SnezanaIPatuljci = 010; // broj 8 (decimalno)
```

(heksadecimalne vrednosti imaju prefiks 0x, a oktalne 0)

2.3 Znakovni podaci (char)

- Znakovni tip podataka smešta se u bajtove, koji se mogu interpretirati i kao celi brojevi

```
char jednoSlovo = 'A'  
jednoSlovo = 65 // ASCII kod slova A
```

- Program:

```
int main() {  
    char jednoSlovo;  
    jednoSlovo = 65;  
    cout << jednoSlovo << endl;  
    jednoSlovo = 'B';  
    cout << jednoSlovo << endl;  
    return 0  
}
```

daje rezultat:

```
A  
B
```



Tip string

- U jeziku C++ postoji tip *literala string* za niz znakova zadanih između dvostrukih navodnika, npr.
`"dobar dan"`
`"ovo je string"`
- Jezik C++ samo koristi *konstante* ovog tipa, ali nema ugrađeni tip podatka *string*, već su nizovi znakova *strukture* podržane preko biblioteke klasa, u kojoj postoji klasa *string*
- Napomena:
 - znakovni tip *wchar_t* koristi se za zapis znakova jezika kao što je kineski, gde 8 bita nije dovoljno za predstavljanje jednog znaka, već se koristi veći broj bita, fiksni ili promenljiv

2.4 Brojevi u pokretnom zarezu (float, double)

- Decimalne vrednosti se mogu zadavati u decimalnoj ili eksponencijalnoj notaciji, npr. isti broj u dva zapisa
 - 123.123
 - 1.23123e2
 - decimalni zapis mora da sadrži decimalnu tačku ili eksponent
- Decimalni broj se može deklarisati kao *float* ili *double*, pri čemu je opseg brojeva tipa *double* za red veličine veći
- Tip *double* se zbog većeg raspona vrednosti koristi češće, a koristi ga i većina matematičkih funkcija u C++ biblioteci

```
float pi = 3.1415;
```

```
float naelektrisanjeElektrona = 1.6e-19;
```

```
double masaSunca = 2e30
```

2.5 Logički tip (bool)

- Koristi se za smeštanje dveju mogućih logičkih vrednosti, istina (*true*) i laž (*false*). Vrednost istina je bilo koja vrednost različita od nule, dok je vrednost laž jednaka nuli. Program...

```
#include <iostream>
using namespace std;
int main() {
    bool b = false; // laž
    cout << "b je" << b << endl;
    b = true;       // istina
    cout << "b je" << b << endl;
    return 0;
}
```

... daje rezultat:

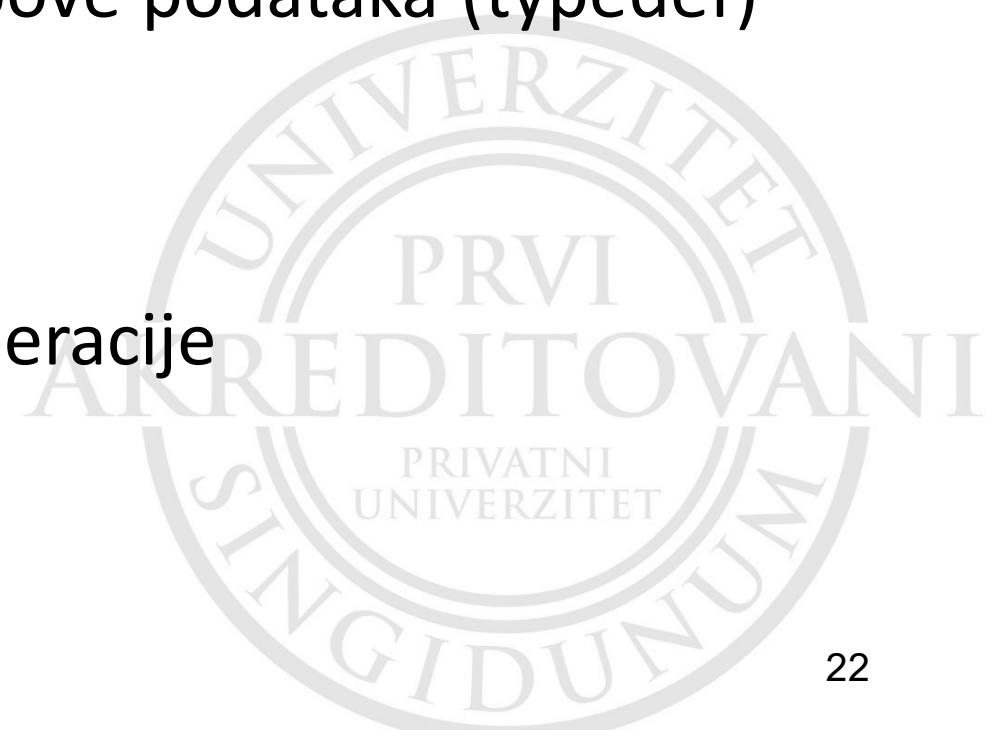
```
b je 0
b je 1
```

3. Operatori i izrazi u jeziku C++

1. Pregled operatora u jeziku C++
2. Evaluacija izraza i prioritet operatora
3. Konverzija tipova podataka (cast)
4. Imenovane konstante

3.1 Pregled operatora u jeziku C++

- Operator dodele vrednosti
- Aritmetički operatori
- Relacioni i logički operatori
- Definisanje sinonima za tipove podataka (typedef)
- Operator sizeof
- Operator nabrajanja (,)
- Osnovne ulazno-izlazne operacije



Operator dodele vrednosti

- Osnovna forma naredbe za dodelu vrednosti je

```
promenljiva = izraz;
```

- Dozvoljena je višestruka dodata ili nizanje operatora, npr.

```
int x, y, z;  
x = y = z = 100;
```

— operator menja *vrednost* objekta, ali *tip* objekta ostaje isti

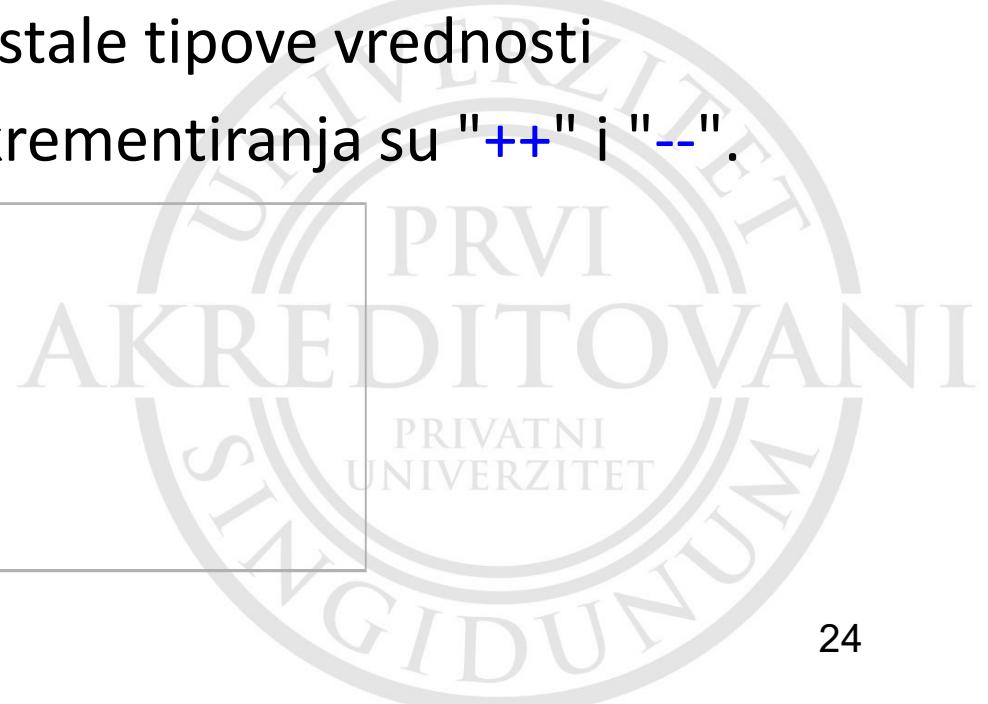
- S leve strane operatora mogu biti samo promenljivi objekti (*lvalues*) , a s desne strane i konstante (*rvalues*), npr.

```
3.1415 = pi; // greška s leve strane  
int i;  
i = i+5;  
int j = 5;
```

Aritmetički operatori

- U jeziku C++ definisani su aritmetički operatori "+", "-", "*" i "/", koji se mogu primeniti na sve numeričke tipove podataka, kao i na podatke tipa `char`
- Deljenje je celobrojno kad se primeni na celobrojni tip podataka. Operator `%` (modul) daje ostatak celobrojnog deljenja i ne primenjuje se na ostale tipove vrednosti
- Operatori inkrementiranja i dekrementiranja su "`++`" i "`--`".

```
int i = 0;  
++i;  
cout << i; // ispisuje 1  
--i;  
cout < i; // ispisuje 0  
<
```



Operatori inkrementa i dekrementa

- Operator inkrementa $x++$ vrši operaciju $x = x+1$, a operator dekrementa $x--$ operaciju $x = x-1$
- Operatori se mogu koristiti *prefiksno* i *postfiksno*, odnosno ispred i iza operanda
- Kada je operand *ispred* promenljive, prvo se vrednost promeni, a zatim se dalje u izrazima koristi nova vrednost
- Kada je operand *iza* promenljive, prvo se postojeća vrednost upotrebi u izrazu, a nakon toga se promeni, npr.

```
int x = 1, y;  
y = ++x; // vrednost y je 2, a x je 2  
y = x--; // vrednost y je 2, a x je 1
```

Provera (mali test): Evaluacija operatora inkrementa i dekrementa

(a) Konačno x kad je promenljiva s obe strane operatora dodele?

```
int x = 1;  
  
x = x++;
```

(b) Koja vrednost će se prikazati na ekranu?

```
#include <iostream>  
  
use namespace std;  
  
int main() {  
  
    int x, a=2, b=4, c=5;  
    x = a-- + b++ - ++c;  
    cout << "x = " << x << endl;  
    return 0  
}
```



Skraćena dodela vrednosti

- Operatori inkrementa/dekrementa mogu se koristiti za skraćeni zapis naredbe dodele vrednosti u obliku naredbe:

```
leva_strana operator = desna_strana
```

što je ekvivalentno naredbi:

```
leva_strana = leva_strana operator desna_strana
```

- Mogu se koristiti sledeći aritmetički i logički operatori:
+, -, *, /, %, <<, >>, &, ^, >
- Ovo svojstvo je preuzeto iz jezika C (efikasniji izvršni kod)
Sledeće tri naredbe su ekvivalentne:

```
x = x + 1; // standardna dodata vrednost  
x++;        // standardni inkrement  
x+=1;       // skraćena dodata vrednost
```

Relacioni i logički operatori

- Relacioni operatori omogućavaju međusobno poređenje različitih vrednosti:

$x > y$ da li je x veće od y ?

$x < y$ da li je x manje od y ?

$x \geq y$ da li je x veće ili jednako y ?

$x \leq y$ da li je x manje ili jednako y ?

$x == y$ da li je x jednako y ?

$x != y$ da li je x različito od y ?

- U jeziku C++ jednostruki znak jednakosti ($=$) predstavlja operator dodele vrednosti, a kao relacioni operator koristi se dvostruka jednakost ($==$)
- Rezultat poređenja može biti **0** (netačno, *false*) ili **1** (*true*)

Primer: Prikaz vrednosti istinitosti

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

int main() {
    int istinaVrednost, lazVrednost, x = 5, y = 10;
    istinaVrednost = x < y;      // 5 < 10
    lazVrednost = x == y; // 5 = 10
    cout << "Istina je:" << istinaVrednost << endl;
    cout << "Laz je:" << lazVrednost << endl;
    return 0;
}
```

Program daje rezultat:

Istina je 1
Laz je 0

Logički operatori

- Logički operatori omogućavaju povezivanje više relacionih izraza u složeniji logički izraz
- Operatori mogu biti (po prioritetu izvršavanja):

!	NOT	<i>logička negacija (unarni)</i>
&&	AND	<i>logičko i (binarni)</i>
	OR	<i>logičko ili (binarni)</i>

- logička negacija menja vrednost operanda *true* u *false* i obrnuto
- logičko i daje rezultat *true* samo ako su oba operanda *true*, inače je rezultat *false*
- logičko ili daje rezultat *true* ako je bar jedan od operanada *true*, a rezultat je *false* je samo ako su oba operanda *false*

Definisanje sinonima za tipove podataka (typedef)

- Jezik C++ omogućava definisanje sopstvenih tipova podataka pomoću deklaracije *typedef*, npr.
`typedef long int VelikiCeoBroj`
- Na taj način se tip **long int** može definisati u naredbama kao
`VelikiCeoBroj brojacSlogova = 0L;`
što je ekvivalentno i može se koristiti istovremeno sa:
`long int brojacSlogova = 0L;`
- Upotreba sinonima
 1. olakšava korišćenje složenih deklaracija i
 2. omogućava *lakše prenošenje* programa na druge platforme, jer se sve deklaracije koje zavise od platforme mogu grupisati i promeniti na jednom mestu

Operator sizeof

- Zapis različitih tipova podataka zavisi od arhitekture računara i samog prevodioca
- Podaci o rasponu vrednosti mogu se naći u zaglavljima, za celobrojne vrednosti u **climits** (npr. INT_MIN i INT_MAX), a za decimalne vrednosti u zaglavlu **cfloat**
- Veličina memorije (u bajtovima) pojedinačnih tipova podataka može se dobiti pomoću funkcije **sizeof(tip)**

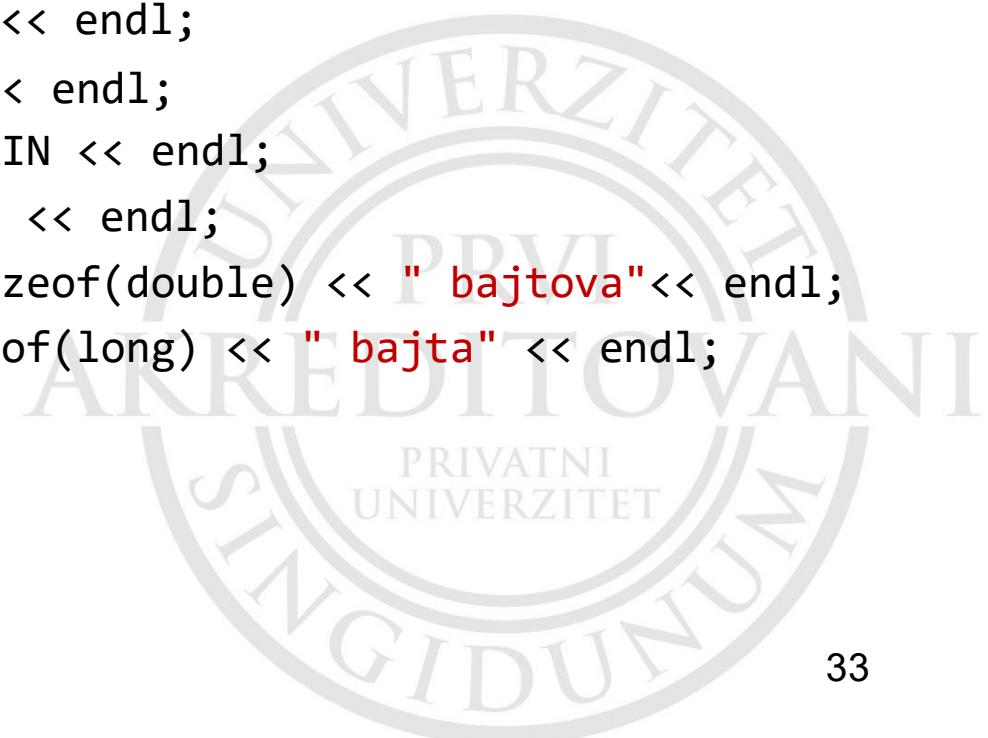
Primer: Upotreba funkcije sizeof

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

int main() {
    cout << "Najmanji int:" << INT_MIN << endl;
    cout << "Najveći int:" << INT_MAX << endl;
    cout << "Najmanji double:" << DBL_MIN << endl;
    cout << "Najveći float:" << FLT_MAX << endl;
    cout << "Tip double zauzima:" << sizeof(double) << " bajtova" << endl;
    cout << "Tip long zauzima:" << sizeof(long) << " bajta" << endl;
    return 0;
}
```

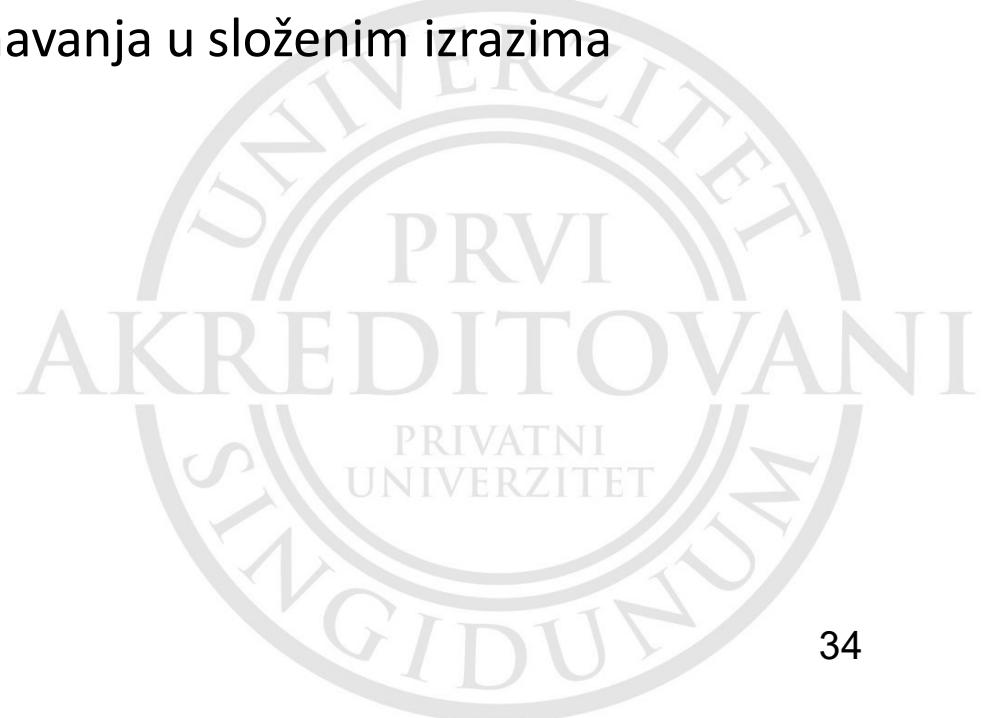
Program daje rezultat:

```
Najmanji int: -2147483648
Najveći int: 2147483648
Najmanji double: 2.22507e-308
Tip double zauzima 8 bajtova
Tip long zauzima 4 bajta
```



Operator nabranja (,)

- Operator nabranja (razdvajanja) omogućava istovremeno računanje niza izraza u jednoj naredbi
- Rezultat izračunavanje je vrednost *poslednjeg* izraza u nizu, računajući s leva u desno
 - **Napomena:** prioritet operatora nabranja je nizak, tako da treba pažljivo razmotriti redosled izračunavanja u složenim izrazima



Primer: Izračunavanje niza izraza

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

int main() {
    long num1=0, num2=0, num3=0, num4=0;
    num4 = (num1=10, num2=20, num3=30); // niz izraza
    cout << "Vrednost niza izraza je vrednost:" <<
        " poslednjeg u nizu: " << num4 << endl;
    return 0;
}
```

Izvršavanje programa daje rezultat:

Vrednost niza izraza je vrednost poslednjeg u nizu: 30

Osnovne ulazno-izlazne operacije

- Ulaz i izlaz podataka sa standardnih ulaznih i izlaznih uređaja (tastatura i ekran) vrši se operatorima umetanja (*insertion*) i izdvajanja (*extraction*)
 - Operator umetanja (`<<`) umeće znakove u izlazni tok (*output stream*) i vrši automatsku konverziju osnovnih tipova podataka u niz znakova
 - Operator izdvajanja (`>>`) izdvaja podatke iz ulaznog toka (od početka linije do Enter) i vrši neophodne konverzije, npr.
- Preciznije formatiranje podataka vrši se posebnim manipulatorskim funkcijama (*Tema 11*)

```
int i;  
cout << "Unesite ceobrojnu vrednost: ";  
cin >> i;  
cout << "Uneli ste vrednost " << i;
```

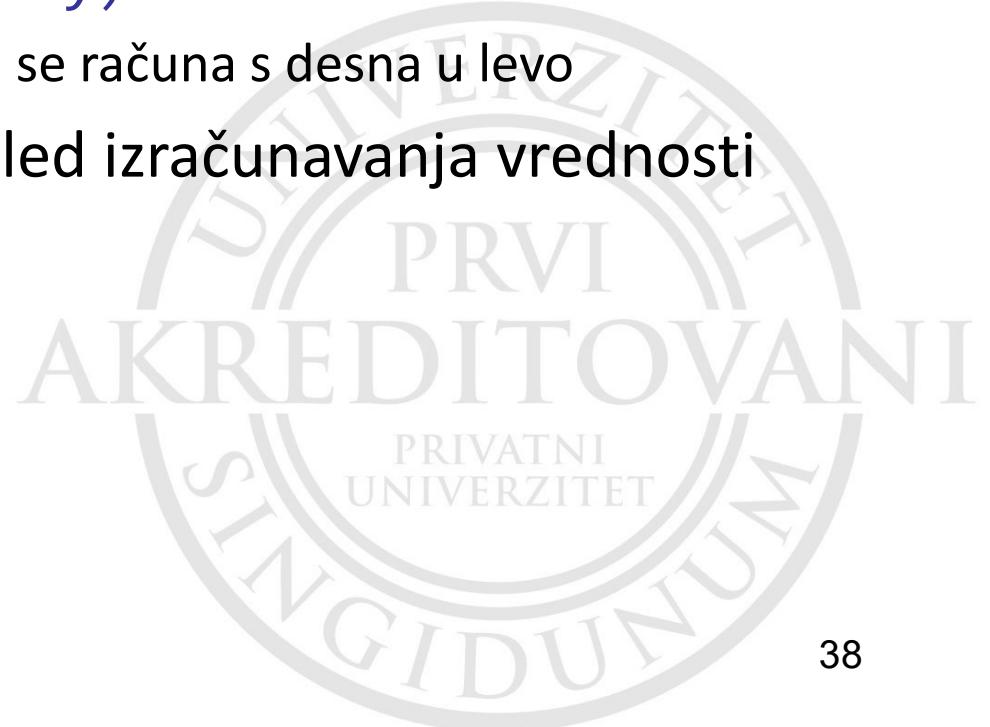
3.2 Evaluacija izraza i prioritet operatora

- Evaluacija izraza
- Prioritet operatora



Evaluacija izraza

- Redosled evaluacije/računanja vrednosti izraza određen je vrstom i prioritetom operatora
- Prefiksni, unarni i operatori dodele vrednosti izvršavaju se s desna u levo, a svi ostali operatori s leva u desno, npr.
 - izraz $x-y-z$ računa se s leva kao $(x-y)-z$
 - u izrazu $x=y=z$ dodela vrednosti $=$ se računa s desna u levo
- Zgrade menjaju osnovni redosled izračunavanja vrednosti izraza



Prioritet operatora (1)

viši prioritet



niži prioritet

Operator	Opis
::	Razrešenje dosega (scope) najviši prioritet
.	Pristup podatku članu
->	Dereferenciranje i pristup podatku članu
[]	Indeksiranje niza
()	Poziv funkcije
++	Inkrementiranje
--	Dekrementiranje
typeid	Identifikacija tipa tokom izvršavanja
const_cast	Ukidanje nepromenljivosti tipa
dynamic_cast	Konverzija tipa tokom izvršavanja
static_cast	Konverzija tipa tokom prevodenja
reinterpret_cast	Neproverena konverzija tipa
!	Logička negacija
~	Bitska negacija
+ (unarni)	Promena znaka
- (unarni)	
* (unarni)	Dereferenciranje pokazivača
& (unarni)	Adresa promenljive
new	Dinamička alokacija memorije
delete	Oslobađanje dinamičke memorije
sizeof	Veličina promenljive ili tipa

Prioritet operatora (2)

viši prioritet



niži prioritet

Operator	Opis
<code>.*</code> (unarni) <code>->*</code>	Pristup podatku članu Dereferenciranje i pristup podatku članu
<code>*</code> <code>/</code> <code>%</code>	Množenje Deljenje Modulo (ostatak deljenja)
<code>+ -</code>	Sabiranje i oduzimanje
<code><< >></code>	Pomeranje za jedan bit uлево ili uдесно
<code>< <= > >=</code>	Relacioni operatori
<code>== !=</code>	Operatori jednakosti
<code>&</code>	Bitsko i (AND)
<code>^</code>	Bitsko islučivo ili (XOR)
<code> </code>	Bitsko ili (OR)
<code>&&</code>	Logičko i
<code> </code>	Logičko ili
<code>:?</code>	Uslovni izraz
<code>= *= /= %= += -= &= ^= >= <<= >>=</code>	Operatori dodele
<code>,</code>	Operator nabranja najniži prioritet

3.3 Konverzija tipova (cast)

- Usaglašavanje tipova objekata s leve i desne strane vrši se prema pravilima konverzije tipova
- Vrednost izraza s desne strane konvertovat će se u vrednost promenljive s leve strane
- Prilikom evaluacije, vrši se konverzija tipova (*type casting*) samih vrednosti u mešovitom izrazu u zajednički kompatibilni tip, npr. tip **int** u **double**, a tip **char** u **int**
- Konverzija u ciljni tip podatka može se izvršiti i eksplisitno, pomoću naredbe: **static_cast <tip> (izraz)**, npr.

```
float pi = 3.1415926;  
cout << static_cast <int> (pi) << endl;
```

3.4 Imenovane konstante

- U programu se određene nepromenjive vrednosti mogu zadati kao simboličke konstante na dva načina:
 - pomoću naredbe `const`
 - pomoću direktive `#define`
- Primer:

```
const double pi = 3.141592653; // u tabeli simbola
#define PI 3.141592653
pi = 2*pi // greška (konstanta s leve strane)!
PI = 2*PI // greška (konstanta s leve strane)!
```

4. Osnovne upravljačke naredbe u jeziku C++

1. Selekcija
2. Iteracija
3. Ostale upravljačke naredbe



4.1 Selekcija

- Osnovne naredbe za selekciju koda koji se izvršava pod određenim uslovima u jeziku C++ su naredbe **if** i **switch**.
- Izabrani kod može biti jedna naredba ili blok naredbi u velikim zagradama, npr.

```
int main() {
    double povrsina;
    cin >> povrsina;
    if (povrsina >= 0) {
        double stranica = sqrt(povrsina);
        ...
    }
    ...
    return 0;
}
```

*Promenljive definisane u okviru nekog bloka naredbi
vidljive su samo u okviru tog bloka*

Naredba *if*

- Osnovni oblik naredbe *if* za selekciju koda koji se izvršava pod određenim uslovima je:

```
if (izraz)
    naredbe;
else
    naredbe;
```

- Izraz vraća vrednost na osnovu koje se bira naredba koja će se izvršiti. Rezultat ne mora biti logički, jer se 0 automatski konvertuje u *false*, a bilo koji drugi rezultat u *true*
- Deo *else* nije obavezan, a izabrane naredbe mogu biti pojedinačna naredba ili blok naredbi

Primer 1: Grananje prema logičkoj ili celobrojnoj vrednosti izraza

```
// Primer grananja
#include <iostream>
using namespace std;
int main() {
    int a;
    cout << "Unesite celi broj: ";
    cin >> a;
    if (a < 0) ← true/false
        cout "Uneti broj je negativan" << endl;
    if (a % 2) ← celi broj
        cout << "Uneti broj je neparan";
    else
        cout << "Uneti broj je paran";
    cout << endl;
    return 0
}
```

Izvršavanje programa:

```
Unesite celi broj: -5
Uneti broj je negativan
Uneti broj je neparan
```



Primer 2: Grananje prema celobrojnoj vrednosti izraza

```
// Primer grananja
#include <iostream>
using namespace std;
int main() {
    int a, b;
    cout << "Unesite deljenik: ";
    cin >> a;
    cout << "Unesite delilac: ";
    cin >> b;
    if (b) ← celi broj
        cout "Rezultat je " << a/b;
    else
        cout "Deljenje nulom nije dozvoljeno.";
    cout << endl;
    return 0;
}
```

Izvršavanje programa:

Unesite deljenik: 50

Unesite delilac: 10

Rezultat je: 10

Unesite deljenik: 50

Unesite delilac: 0

Deljenje nulom nije dozvoljeno.

Primer 3: Višestruka selekcija *(if-else-if)*

```
// Primer računanje diskriminante kvadratne jednačine
#include <iostream>
using namespace std;
int main() {
    float a, b, c;
    cout << "Unesite koeficijente kvadratne jednacine: " << endl;
    cin >> a >> b >> c;
    float diskriminanta = b*b - 4*a*c;
    cout << "Kvadratna jednacina ima ";
    if (diskriminanta > 0)
        cout << "dva realna korena.";
    else if (diskriminanta < 0)
        cout << "dva kompleksna korena.";
    else
        cout << "dvostruki realni koren.";
    cout << endl;
    return 0;
}
```

Izvršavanje programa:

Unesite koeficijente kvadratne jednačine:

-1 2 -1

Kvadratna jednačina ima dvostruki realni koren.



Naredba switch

- Osnovni oblik narebe **switch** za višestruko grananje prema vrednosti nekog izraza je:

```
switch (izraz) {  
    case (konstanta1):  
        niz naredbi;  
        break; ←  
    case (konstanta2):  
        niz naredbi;  
        break;  
    ...  
    default:  
        niz naredbi;  
}
```

*nije obavezna, ali je efikasnije
prekinuti dalje ispitivanje*

Operator selekcije

- Posebna vrsta selekcije je upotreba *operatora selekcije*

uslov ? vrednost₁ : vrednost₂

- Korišćenjem ovog operatora naredba

if (x >= 0) y = x;

else y = -x;

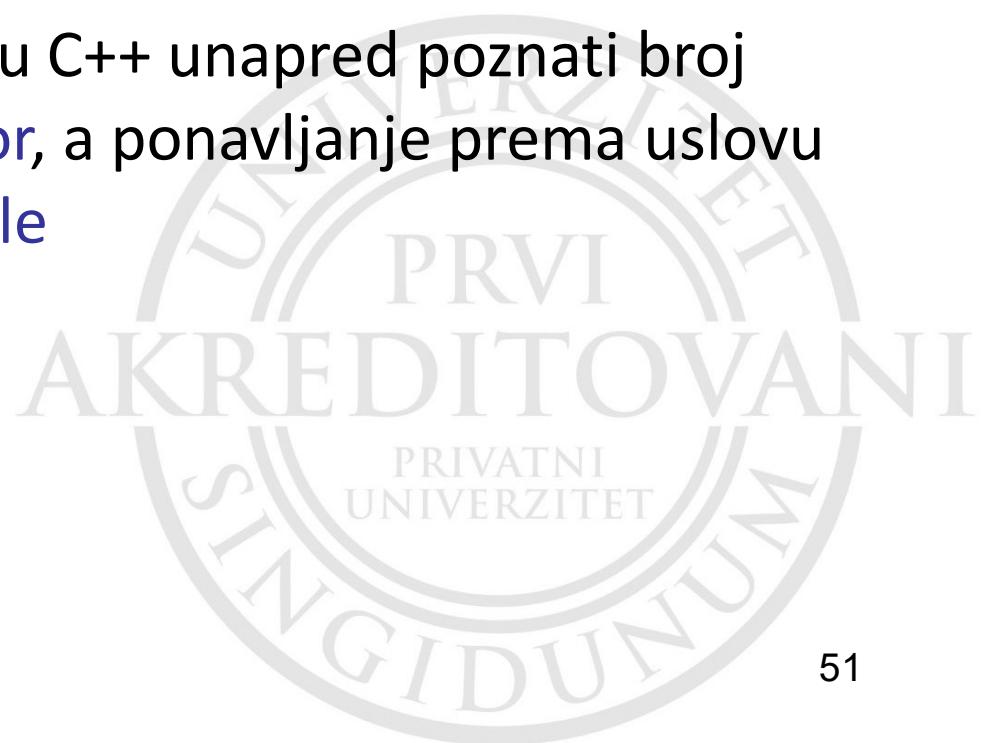
može se kraće napisati kao:

y = x >= 0 ? x : -x;



4.2 Iteracija

- U proceduralnim programskim jezicima ponavljanje se može realizovati u odnosu na uslov okončanja ponavljanja (petlje)
 - unapred određen broj puta
 - prema logičkom uslovu, koji se proverava pre početka ponavljanja ili nakon svakog ponavljanja
- Ponavljenje delova koda u jeziku C++ unapred poznati broj puta vrši se pomoću naredbe **for**, a ponavljanje prema uslovu pomoću naredbi **while** i **do-while**



Naredba *for*

- Sintaksa naredbe *for* je
$$\text{for } (\text{inicijalizacija}; \text{ izraz}; \text{ inkrement}) \text{ blok naredbi};$$
- Inicijalizacija je naredba kojom se postavlja početna vrednost promenljive koja određuje broj izvršavanja petlje
 - ova promenljiva je često potrebna samo kao brojač, pa se deklariše *unutar* definicije for petlje, tako da izvan petlje nije definisana
- Izraz je uslov koji na svakom koraku određuje da li će ponavljanje biti izvršeno
 - mora biti *true*, inače se izvršavanje programa nastavlja od prve sledeće naredbe *iza* petlje *for*
- Inkrement je definicija operacije promene promenljive koja upravlja ponavljanjem

Naredba *while*

- Naredba *while* ima sintaksu

```
while (izraz) blok naredbi;
```

- Npr. za Euklidov algoritam se ne zna unapred broj ponavljanja

```
#include <iostream>
using namespace std;
int main() {
    int x, y;
    cout << "Unesite brojeve ciji NZD trazite: " << endl;
    cin >> x >> y;
    while (x != y)
        if (x > y) x = x-y;
        else y = y-x;
    cout << "NZD=" << x << endl;
    return 0;
}
```



Naredba *do-while*

- Naredba *do-while* ima sintaksu

```
do
    blok naredbi
    while (izraz);
```

- Primer: naknadno odlučivanje o nastavku programa:

```
#include <iostream>
using namespace std;
int main() {
    const int TAJNI_BROJ = 15;
    int broj = 0;
    do {
        cout << "Unesite tajni broj: " << endl;
        cin >> broj;
    } while (broj != TAJNI_BROJ);
    cout << "Pogodili ste tajni broj!"<< endl;
    return 0;
}
```



4.3 Ostale upravljačke naredbe

- Naredba bezuslovnog skoka (*goto*)
- Naredbe *break* i *continue*



Naredba bezuslovnog skoka (*goto*)

- Naredba bezuslovnog skoka na lokaciju u programskom kodu koja je navedena iza reči *goto*, npr.

```
if (imenilac == 0) goto deljenjeNulom;  
// naredbe koje se preskaču ...  
deljenjeNulom:  
    cout << "Deljenje nulom nije dozvoljeno"
```

- Naredba odgovara mašinskoj instrukciji bezuslovnog skoka
- U savremenom programiranju praktično se *ne koristi*, jer čini programski kod manje čitljivim i teškim za održavanje
- U savremenim programskim jezicima uvek postoji drugi, elegantniji način realizacije istog algoritma *bez* naredbe *goto*

Naredba *break*

- Naredba **break** se koristi za prekid izvršavanja petlji, nakon čega se program nastavlja na prvoj sledećoj naredbi *iza petlje*
- Npr. sledeća petlja se prekida kada kontrolna promenljiva x dostigne vrednost 3:
 - Kod ugnježdenih petlji, *break* prekida samo unutrašnju petlju
 - Naredba se koristi za prekid petlje u posebnim situacijama, kao što je npr. prekid beskonačne petlje

```
for (int x=10; x > 0; x--)  
    if (x == 3) {  
        cout << "Odbrojavanje je završeno.";  
        break;  
    }
```

Naredba *continue*

- Naredba **continue** se koristi za prekid izvršavanja samo *jedne iteracije* petlje
- Program se nastavlja od provere uslova petlje za izvršenje sledeće iteracije i eventualno prelazi na *sledeću iteraciju*

```
for (int x=0; x <= 100; x++) {  
    if (x % 2) // preskače neparne brojeve  
        continue;  
    else  
        cout << x << endl;  
}
```



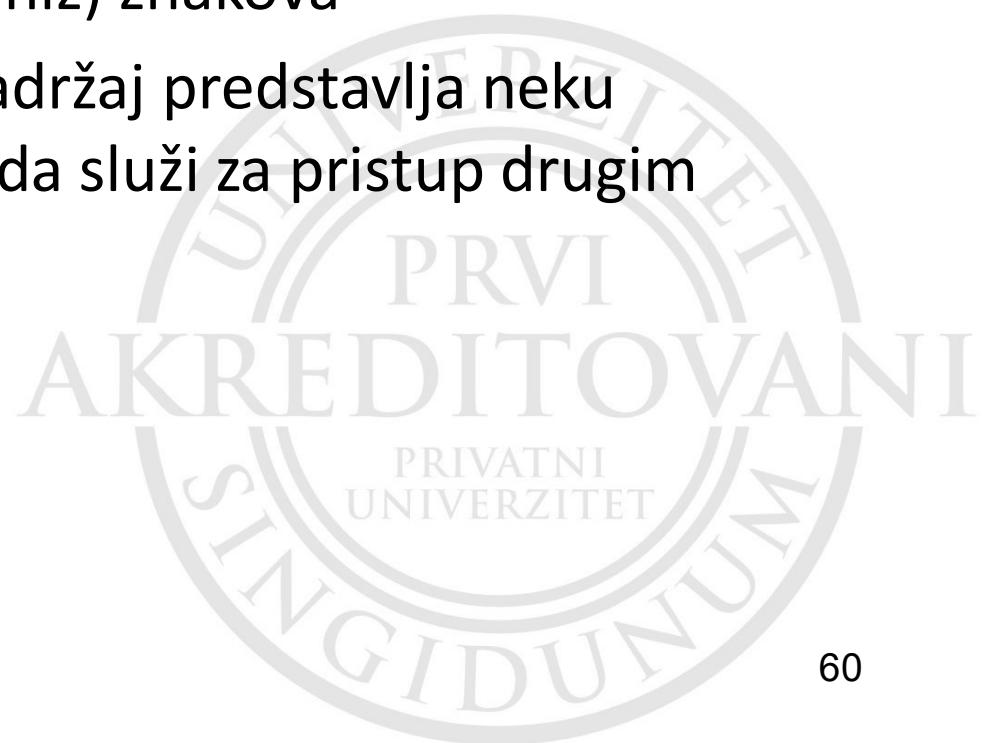
5. Strukture podataka u jeziku C++

1. Polja (nizovi)
2. Stringovi
3. Polja stringova
4. Pokazivači
5. Korisnički definisane strukture
6. Povezane liste



Strukture podataka u jeziku C++

- Polja i stringovi predstavljaju skup promenljivih u memoriji kojima se pristupa pomoću zajedničkog imena
 - mogu imati jednu ili više dimenzija
- Tip string ne predstavlja osnovni tip podataka u jeziku C++, već jednodimenzionalno polje (niz) znakova
- Pokazivači su promenljive čiji sadržaj predstavlja neku memorijsku adresu, koja može da služi za pristup drugim objektima u memoriji



5.1 Polja (nizovi)

- Skup povezanih promenljivih istog tipa deklariše se naredbom
 $tip\ ime_polja\ [dimenzija_1]\ [dimenzija_2]\ ..\ [dimenzija_N];$
- Jednodimenzionalno polje ili vektor je npr.
`int primer [10];`
- Višedimenzionalno polje je npr.
`int primer2D[10][20];`
- Pristup elementima polja vrši se pomoću *indeksa*, npr.
`primer[10]`
`primer2D[3][5]`
- U jeziku C++ indeksi počinju od **0**, tako da polje od 10 elemenata ima indekse od **0** do **9**, npr. `primer[0]..primer[9]`

Primer: Deklarisanje i inicijalizacija polja

```
const int BROJ_REDJOVA = 3;  
const int BROJ_KOLONA = 4;  
// Deklaracija polja celih brojeva od 3 reda i 4 kolone  
int brojevi[BROJ_REDJOVA][BROJ_KOLONA];  
// Prvi indeks predstavlja red, a drugi kolonu  
for (int red=0; red < BROJ_REDJOVA; red++)  
    for (int kolona=0; kolona < BROJ_KOLONA ; kolona++){  
        brojevi[red][kolona]= 4*red + kolona + 1  
    }  
}
```

Šta predstavlja ovaj izraz?

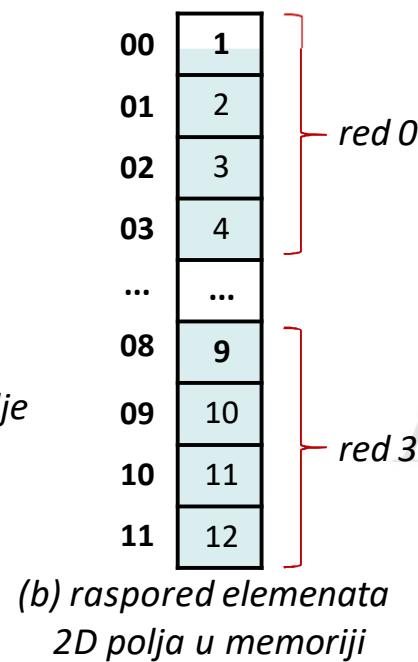
- Vrednost promenljive `brojevi[0][0]` je 1, `brojevi[0][1]` je 2, `brojevi[0][2]` je 3, a npr. `brojevi[2][3]` je 12

Predstavljanje polja u memoriji

- Polja se u memoriji predstavljaju obrnuto od redosleda indeksa, tako da se poslednji indeks menja najbrže
- Npr. dvodimenzionalno polje (a) u memoriji se predstavlja tako da su *redovi* u sukcesivnim memorijskim lokacijama (b)

		kolona			
		0	1	2	3
red	0	1	2	3	4
	1	5	6	7	8
2	9	10	11	12	

(a) dvodimenzionalno polje



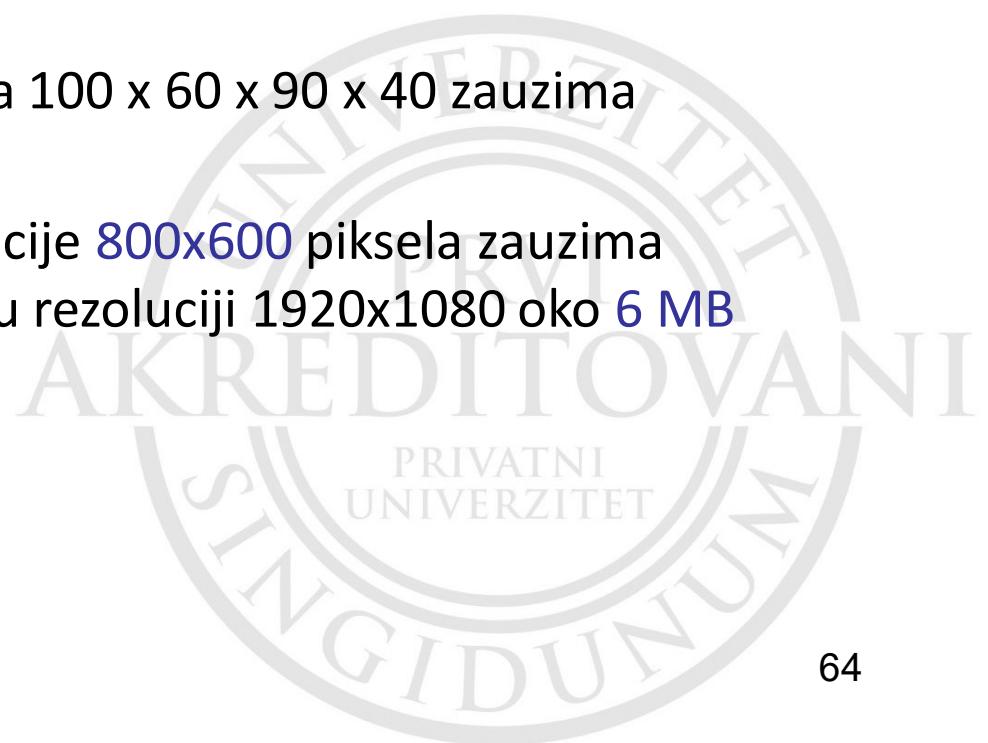
Važno za upotrebu programskih *biblioteka*:

A. predstavljanje polja prvo po *redovima*:
C, *C++*, *Objective-C* (samo za *C*-polja),
Pascal, *C#*

B. predstavljanje polja prvo po *kolonama* :
Fortran, *OpenGL* , *MATLAB/GNU Octave*,
Objective-C, *R* (jezici *Java*, *Javascript*,
Python i *PHP* koriste pokazivače)

Predstavljanje polja u memoriji

- Memorijski prostor za predstavljanje polja određuje se u toku prevođenja kao
$$\text{dimenzija}_1 \times \text{dimenzija}_2 \times \dots \times \text{dimenzija}_N \times \text{sizeof}(\text{tip elementa})$$
- Zauzeće memorije raste progresivno s brojem i veličinom dimenzija, npr.
 - četvorodimezionalno polje znakova $100 \times 60 \times 90 \times 40$ zauzima 21.600.000 bajtova (oko 20 MB)
 - rasterska slika u RGB koloru rezolucije 800x600 piksela zauzima 480.000×24 bita ili oko 1,4 MB, a u rezoluciji 1920x1080 oko 6 MB



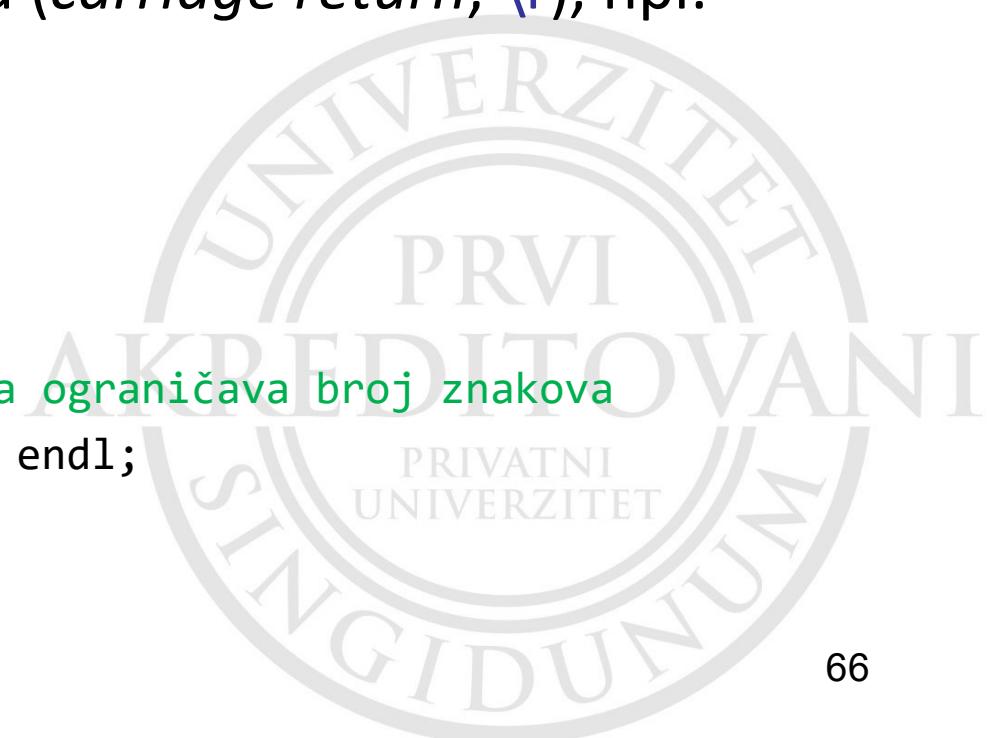
5.2 Stringovi

- Stringovi su jednodimenzionalna polja (nizovi) znakova
- U jeziku C++ koriste se *dva načina* predstavljanja nizova znakova:
 1. C string, niz znakova koji se terminira nul-znakom \0 (*null terminated string*), nasleđen iz jezika C, koji omogućava efikasno predstavljanje i detaljnu kontrolu programera nad operacijama sa stringovima
 2. Klasa string iz biblioteke klasa jezika C++
- String ili niz znakova koji završava nulom deklariše se tako da dimenzija niza bude *za jedan veća* od najveće dužine stringa koji će se koristiti
 - npr. za string str dužine 10 znakova ispravna deklaracija je
`char str[11]`

Učitavanje stringa s tastature

- Niz znakova se može direktno učitati naredbom `cin`, samo se mora voditi računa da naredba `string` čita do prve "beline" (*whitespace*: prazno mesto, tabulator ili znak za novi red)
- Bibliotečna funkcija `gets()` omogućava čitanje svih znakova stringa sve do znaka za novi red (*carriage return*, `\r`), npr.

```
#include <iostream>
using namespace std;
int main() {
    char str[81];
    cout << "Unesite string :";
    gets_s(str, 80); // nova verzija ograničava broj znakova
    cout << "Uneli ste: " << str << endl;
    return 0;
}
```



Funkcije za rad sa stringovima

- Najčešće korištene funkcije za rad sa stringovima u biblioteci funkcija jezika C++ su:

Funkcija	Opis
<code>strcpy(s1, s2)</code> <code>strcpy_s()</code> <code>wcsncpy()</code>	Kopira string s2 u string s1. Niz s1 mora da bude dovoljno dugačak, jer će se u suprotnom biti prekoračena granica niza.
<code>strcat(s1, s2)</code> <code>strcat_s()</code> <code>wcsconcat()</code>	Dodaje s2 na kraj s1; s2 ostaje neizmenjen. s1 mora da bude dovoljno dugačak da u njega stane sadrži prvobitni sadržaj i ceo niz s2.
<code>strcmp(s1, s2)</code> <code>wcsncmp()</code>	Poredi stringove s1 i s2; ako su jednaki, vraća 0, ako je $s1 > s2$ (po leksikografskom redu) vraća 1, u suprotnom vraća negativan broj.
<code>strlen(s)</code>	Vraća dužinu stringa s.

Vraća `0 == false`
kad su stringovi
jednaki

Funkcije za rad sa pojedinačnim znakovima

- U zaglavlju `<cctype>` standardne biblioteke jezika C++ definisane su funkcije za rad sa znakovima, npr.

Funkcija	Opis
<code>toupper(c)</code>	prevodi znak u veliko slovo (uppercase)
<code>tolower(c)</code>	prevodi znak u malo slovo (lowercase)
<code>isupper(c)</code>	true ako je znak veliko slovo
<code>islower(c)</code>	false ako je znak veliko slovo
<code>isalnum(c)</code>	true ako je znak alfanumerički
<code>isdigit(c)</code>	true ako je znak cifra
<code>isspace(c)</code>	true ako je znak razmak

Primer: Konverzija stringa u velika slova

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

int main() {
    char str[80];
    strcpy(str, "Ovo je test!");
    // Konverzija znakova stringa u velika slova
    for (int i=0; str[i], i++)
        str[i] = toupper(str[i]);
    cout << str << endl;
    return 0;
}
```

Izvršavanje programa:

OVO JE TEST!

Ekvivalentno `str[i] == 0` (kraj stringa)
odnosno `false`

Inicijalizacija stringova

- Inicijalizacija polja vrši se naredbom oblika
$$\text{tip } \textit{ime_polja} \text{ [dimenzija]} = \{\textit{skup_vrednosti}\}$$
- Skup vrednosti je niz vrednosti odgovarajućeg tipa odvojenih zarezima, npr. {10, 20, 30, 40}
- Kada se string istovremeno deklariše i inicijalizuje, dimenzije se mogu izostaviti, jer ih prevodilac može odrediti automatski
- Dimenzije stringova se mogu izostaviti, npr. umesto
`char str[4] = "C++";`
može se napisati (sistem dodaje oznaku `\0` na kraju stringa)
`char str[] = "C++";`

5.3 Polja stringova

- **Polje stringova** je poseban oblik dvodimezionalnog niza, koji se često koristi za različite operacije pretraživanja
- U deklaraciji polja, prvi indeks je broj nizova, a drugi maksimalna dužina niza (uključujući oznaku kraja), npr.
`char daniUnedelji[7][11] = {"Ponedeljak", "Utorak",
"Sreda", "Cetvrtak", "Petak", "Subota", "Nedelja"};`
- Stringovima se pristupa pomoću jednog, a elementima stringa pomoću dva indeksa, npr.

```
daniUnedelji[3]      // Cetvrtak
```

```
daniUnedelji[3][2]   // Prvi znak "t" u stringu Cetvrtak
```

Primer: Telefonski imenik

```
#include <iostream>
using namespace std;
int main() {
    int i; // brojač, dostupan izvan petlje
    char str[80];
    // Definisanje niza od 10 stringova dužine do 79 znakova
    char brojevi[10][80] = {
        "Pera", "065-234-1123",
        "Deki", "063-1123400",
        "Sandra", "062-3256343",
        "Laza", "061-3453453"
    }
    cout << "Unesite ime: ";
    cin >> str
    for (i=0; i < 10; i+=2) {
        if (!strcmp(brojevi[i], str)) {
            cout << "Broj je: " << brojevi[i+1] << endl;
            break;
        }
    }
    if (i = 10)
        cout << "Broj nije pronadjen." << endl;
    return 0;
}
```

Izvršavanje programa:

Unesite ime: Deki
Broj je: 063-1123400

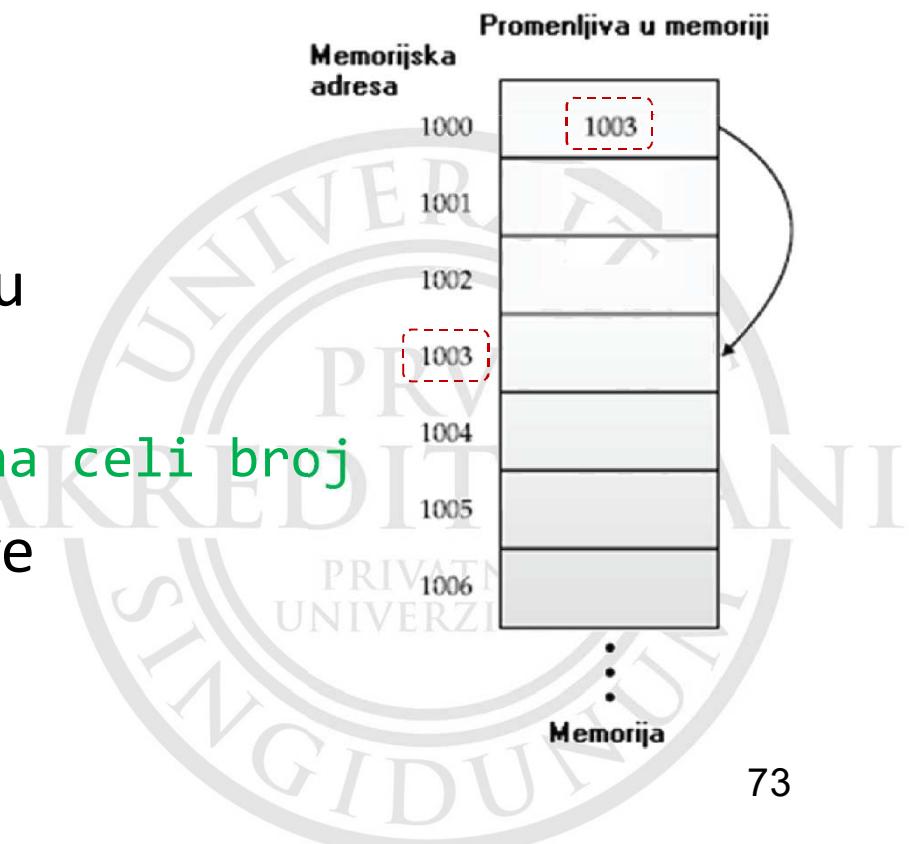
Unesite ime: Filip
Broj nije pronadjen.



Funkcija `strcmp` poređi stringove i vraća 0
odносно `false` kad su stringovi isti

5.4 Pokazivači

- Pokazivač (*pointer*) je promenljiva koja sadrži memorijsku adresu nekog objekta ("pokazuje" na objekt)
- Adresa nije običan celi broj, pa se deklariše na poseban način:
`tip *promenljiva`
- Tip predstavlja *tip podatka* na koje pokazivač pokazuje, npr. pokazivač `intp` na promenljivu tipa `int` deklariše se kao
`int *intp // intp pokazivač na celi broj`
(zvezdica ispred imena promenljive označava pokazivač)

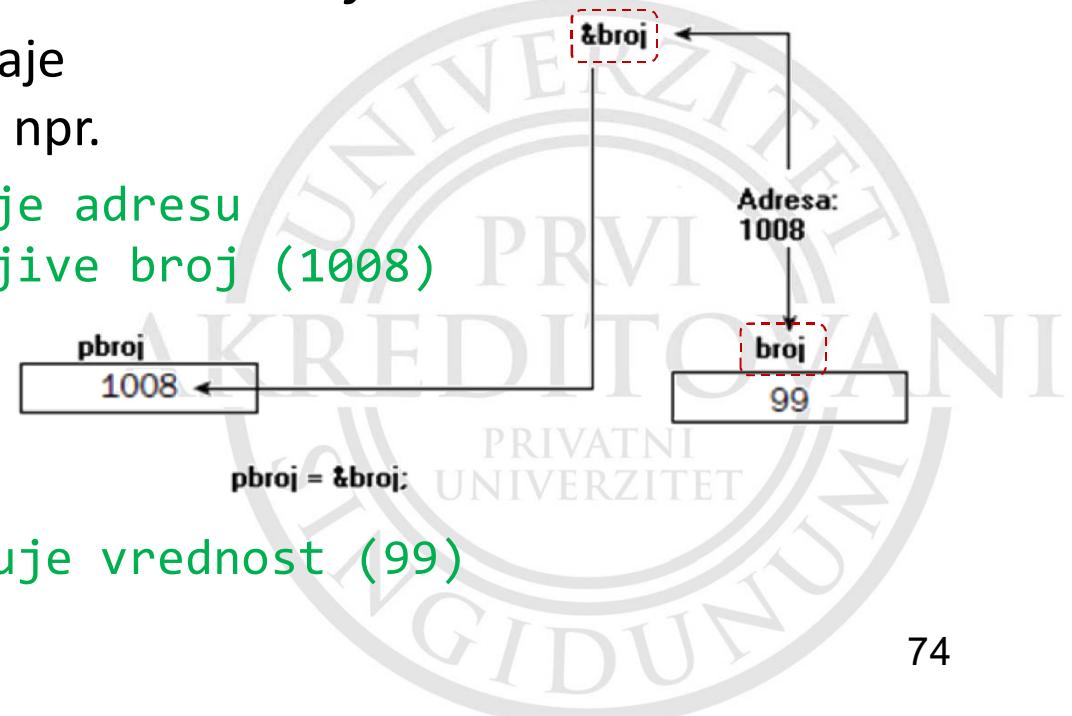


Operacije nad pokazivačima

- Da se izbegnu opasne reference usled neinicijalizovanih pokazivača, usvojeno je da se inicijalizuju na vrednost **NUL** (nula), sa značenjem da pokazivač ne pokazuje ni u šta
- Operatori koji omogućavaju **operacije nad pokazivačima** su unarni operatori *adresiranja &* i *indirekcije **
 - operator *adresiranja &* (slika) daje *vrednost adrese* nekog objekta, npr.

```
pbroj = &broj // dodeljuje adresu
// promenljive broj (1008)
```
 - operator *indirekcije ** daje *vrednost promenljive* koja se adresira

```
vredn = *pbroj // dodeljuje vrednost (99)
```



Osnovni tip pokazivača i dodela vrednosti promenljivoj

- Osnovni tip pokazivača (tip promenljive na koju pokazuje) služi za određivanje broja bajtova koji će se upotrebiti u dodeli vrednosti promenljivoj pomoću pokazivača, npr.

```
int *p;    // pokazivač na tip int
double f;

p = &f;    // greška, ne može da pokazuje na tip double
```

- Napomena: dodata i provera NULL vrednosti

```
p = NULL; // pokazivač koji ne pokazuje ni u šta
p = 0;     // ista vrednost kao NULL
if (!p)
    cout << "Pokazivac ne pokazuje ni u sta" << endl;
```

Dodela vrednosti promenljivoj i pokazivačka aritmetika

- Pokazivač se može koristiti za dodelu vrednosti promenljivoj čiju adresu poznajemo, npr. inkrementiranje oblika (`*p++`)

```
int *pbroj = NULL;  
int broj = 10;  
pbroj = & broj; // pokazivač na promenljivu broj (=10)  
(*pbroj)++; // inkrementiranje preko pokazivača (=11)
```

- Pokazivači se mogu koristiti u većini izraza u jeziku C++, ali se koriste samo operatori `+`, `-`, `++` i `--`
- *Rezultat ovih operacija zavisi od osnovnog tipa pokazivača, zbog različitog zauzeća memorije*

Pokazivači i polja

- Elementima polja se može pristupiti na dva načina: pomoću indeksa i (ponekad efikasnije) pomoću pokazivača, npr.

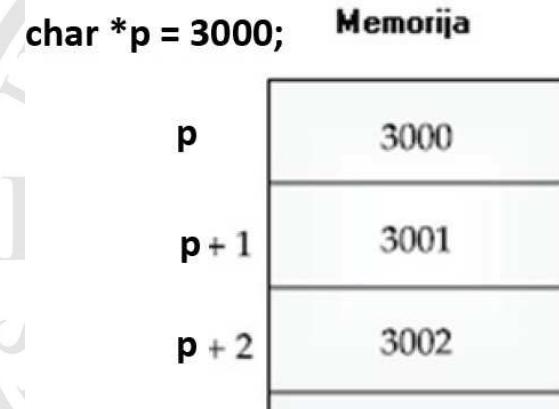
```
char str[80];
```

```
char *p;
```

```
p = str; // dodela adrese prvog znaka str ili str[0]
```

Promenljiva p je pokazivač na znakove, a prva dodeljena vrednost pokazuje na str[0]

- Rad s pokazivačima može biti efikasniji zbog manjeg broja mašinskih instrukcija kad se elementima pristupa po redosledu u memoriji: p, p+1, ...*



Primer: Isti program na dva načina

```
// Pristup pomoću indeksa
#include <iostream>
using namespace std;

int main() {
    char str[] = "Ovo Je Test";
    for (int i=0; str[i]; i++) {
        if (isupper(str[i]))
            str[i] = tolower(str[i]);
        else if (islower(str[i]))
            str[i] = toupper(str[i]);
    }
    cout << str << endl;
    return 0;
}
```

Izvršavanje programa:
oV0 jE tEST

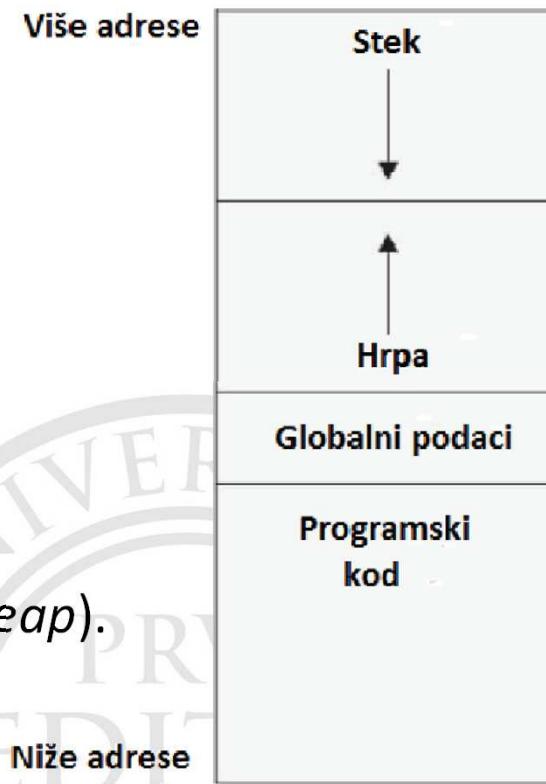
```
// Pristup pomoću pokazivača
#include <iostream>
using namespace std;

int main() {
    char str[] = "Ovo Je Test";
    char *p = str; // *p za str[i]
    while (*p) {
        if (isupper(*p))
            *p = tolower(*p);
        else if (islower(*p))
            *p = toupper(*p);
        p++; // sledeći element str
    }
    cout << str << endl;
    return 0;
}
```

Dinamička alokacija memorije

- Struktura memorije programa
 - Statičke lokalne promenljive i promenljive kreirane izvan svih funkcija su u zoni *globalnih* podataka
 - Lokalne promenljive, koje se kreiraju u bloku naredbi su u zoni koja se koristi kao stek (*stack*) i uništavaju se nakon izlaska iz bloka
 - Dinamički kreirane promenljive (po potrebi), alociraju se u posebnom delu memorije (hrpa, *heap*). Koriste se operatori *new* i *delete*, npr.

```
int *p = new int;
int *q = new int[100];
delete p;
delete [] q;
```



5.5 Korisnički definisane strukture

- Strukture podataka koje sadrže podatke različitih tipova mogu se definisati naredbom **struct**

```
struct naziv { naziv_polja tip; ... };
```

- Stvarni objekti najčešće imaju svojstva različitih tipova, npr.

```
struct Knjiga {  
    char naslov[80];  
    char autor[80];  
    char izdavac;  
    int godina;  
}
```

- Svojstva strukture su **polja** (*fields, members*) i mogu biti bilo kog tipa osim istog onog koji se definiše naredbom **struct**



Upotreba korisničkih struktura

- Promenljiva ovakvog tipa deklariše se na uobičajen način
`Knjiga roman;`
- Inicijalizacija se vrši dodelom skupa vrednosti odgovarajućih tipova:
`Knjiga roman {
 "Na Drini cuprija",
 "Ivo Andric",
 "Zavod za udžbenike",
 2009
}`
- Upotreba promenljive novog tipa vrši se pomoću *operatora selekcije*, npr.

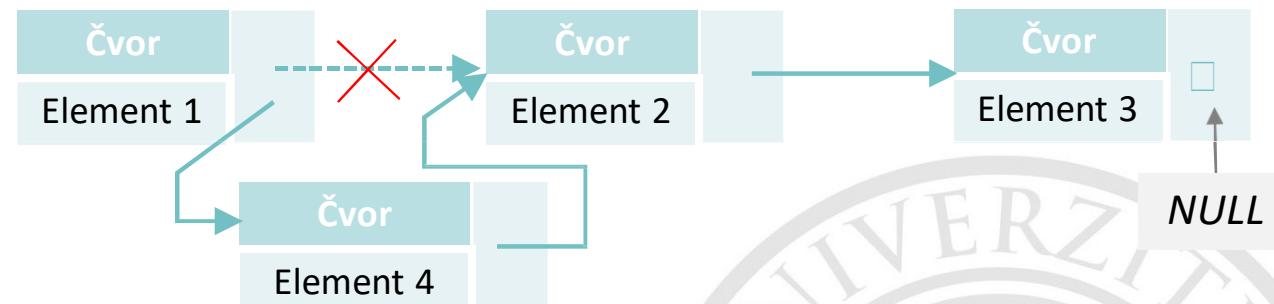
```
roman.godina += 2; // novije izdanje
```



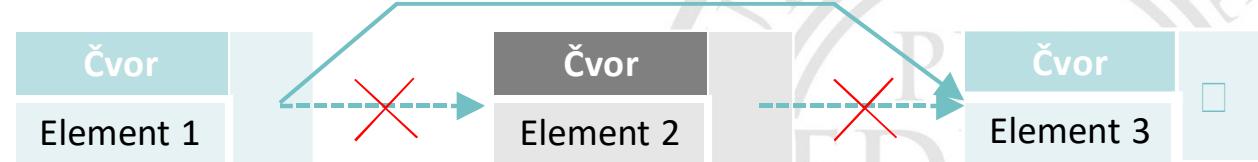
5.6 Povezane liste

- Povezana lista (*linked list*) je struktura koja se sastoji od niza međusobno povezanih elemenata (*nodes*), tako da je dodavanje i uklanjanje elemenata iz liste efikasno

- *dodavanje*



- *uklanjanje*



- Lista može biti jednostruko ili dvostruko povezana
 - može da sadrži jedan pokazivač na sledeći element ili dva pokazivača, na sledeći i prethodni element

Definisanje povezane liste

- Lista se može definisati kao korisnička struktura podataka, koja osim podataka sadrži i pokazivače na sledeći i/ili prethodni element liste, jer pokazivač *može* da pokazuje na strukturu *istog* tipa:

```
struct ElementLista {  
    // definicija elementa strukture  
    ...  
    ElementLista *sledeci;    // pokazivač na element  
    ElementLista *prethodni;  // pokazivač na element  
};
```

- Povezana lista se može upotrebiti za implementaciju drugih struktura podataka, npr. stek (*stack*) i red (*queue*)

6. Primeri programa

1. Kreditni kalkulator u jeziku C++
(bez upotrebe korisničkih funkcija)
2. Sortiranje polja: Selection sort



6.1 Kreditni kalkulator u jeziku C++

- Mesečna rata otplate kredita zavisi od iznosa zajma, mesečne kamate i broja otplatnih rata, koji se uprošćeno može izraziti brojem godina otplate
- Izraz za računanje mesečne rate je [8]

$$rata = \frac{iznos * kamata}{1 - (1 + kamata)^{-n}} = \frac{iznos * kamata}{1 - \frac{1}{(1 + kamata)^n}}$$

gde su:

rata - iznos mesečne rate

kamata - mesečna kamata (godišnja/12)

iznos - iznos kredita

n - broj otplatnih rata kredita (broj godina*12)

Kreditni kalkulator u jeziku C++ (1/2)

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

int main() {

    // Godisnja kamata
    cout << "Unesi godisnju kamatu (decimalni broj): ";
    double godisnjaKamata;
    cin >> godisnjaKamata;

    // Mesecna kamata
    double mesecnaKamata = godisnjaKamata / 1200;

    // Otplatni period u godinama
    cout << "Unesi broj godina oplate zajma (celi broj): ";
    int brojGodina;
    cin >> brojGodina;
```



Kreditni kalkulator u jeziku C++ (2/2)

```
// Iznos zajma
cout << "Unesi iznos zajma (decimalni broj): ";
double iznosZajma;
cin >> iznosZajma;

// Racunanje mesecne rate
double mesecnaRata = iznosZajma * mesecnaKamata /
                     (1 - 1/pow(1 + mesecnaKamata, brojGodina * 12));
double ukupnoVraceno = mesecnaRata * brojGodina * 12;

// Formatiranje izlaza na dve decimale
mesecnaRata = static_cast<int>(mesecnaRata * 100) / 100.0;
ukupnoVraceno = static_cast<int>(ukupnoVraceno * 100) / 100.0;

// Prikaz rezultata
cout << "Mesecna rata " << mesecnaRata <<
    "\nUkupno se otplati " << ukupnoVraceno << endl;
return 0;
}
```

```
Unesi godisnju kamatu (decimalni broj): 4.5
Unesi broj godina otplate zajma (celi broj): 20
Unesi iznos zajma (decimalni broj): 50000
Mesecna rata 316.32
Ukupno se otplati 75917.9
```

6.2 Sortiranje polja: Selection sort

- Ako se sortiranje vrši u rastućem redosledu, metod pronalazi najmanji element u polju i menja ga s prvim elementom
 - sortiranje u opadajućem redosledu traži najveći element
- Postupak pronalaženja najmanjeg elementa i zamene s prvim ponavlja se za ostatak polja, sve dok ne preostane samo jedan element. Opšti oblik metoda je

```
for (i=0, i<broj_elemenata, i++)
    // Izabere se najmanji element polja, od "i+1" do kraja
    // Ako je potrebno, zamene se najmanji i element "i"
```
- Na kraju svake iteracije element i je na konačnoj poziciji. Sledeća iteracija se primenjuje na ostatak polja od $i+1$ do kraja



Program

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

int main () {

    // Sortiranje polja brojeva metodom zamene

    int polje [10] = {33,100,4,55,66,88,7,800,11,0};
    int brojElemenata = 10;

    for (int i=0; i<brojElemenata-1; i++) {
        int currentMin      = polje[i];
        int currentMinInd   = i;
        // Pronadje se najmanji element od polje[i] do kraja
        for (int j=i+1; j<brojElemenata; j++) {    // Selekcija
            if (currentMin > polje[j]) {
                currentMin      = polje[j];
                currentMinInd   = j;
            }
        }
        // Zamene se polje[i] i polje[currentMinIndex], po potrebi
        if (currentMinInd != i) {
            polje[currentMinInd] = polje[i];
            polje[i] = currentMin;
        }
    }
    for (int i=0; i<brojElemenata; i++)
        cout << polje [i] << " ";
    cout << endl;

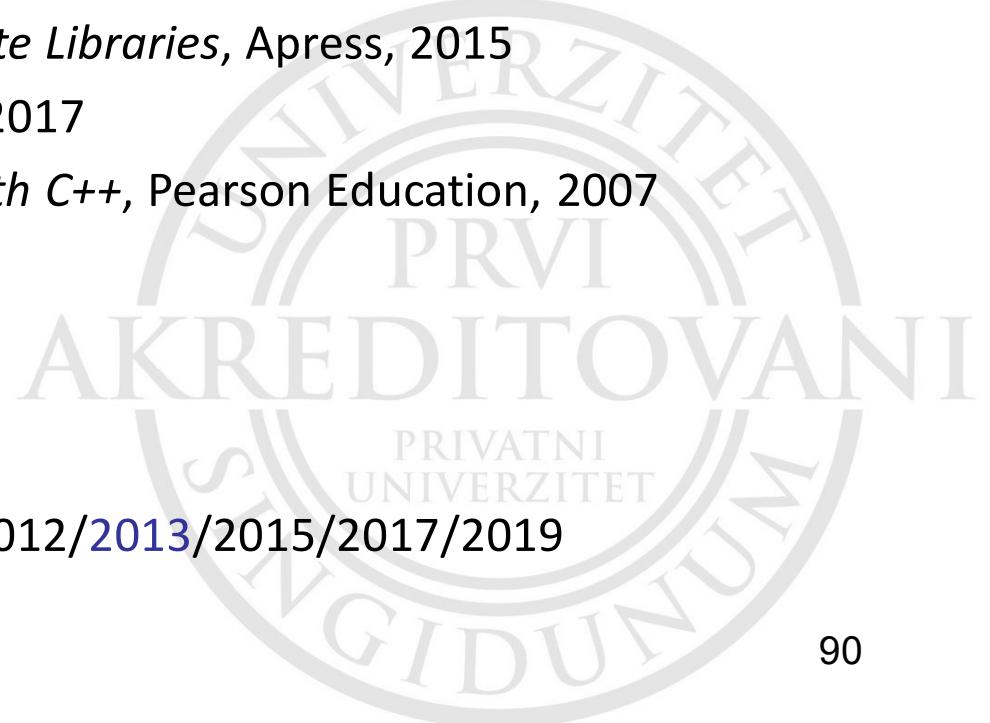
    return 0;
}
```

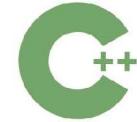


0 4 7 11 33 55 66 88 100 800

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Galowitz J., *C++ 17 STL Cookbook*, Packt, 2017
8. Liang D., *Introduction to Programming with C++*, Pearson Education, 2007
9. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learnCPP.com/>
 - <http://www.stroustrup.com/>
10. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*





Tema 03

Funkcije, prenos parametara i dinamička alokacija memorije

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Funkcije u jeziku C++
3. Oblast važenja promenljivih
4. Prostori imena (imenici)
5. Prenos argumenata funkcije
6. Prototipovi i preklapanje funkcija
7. Rekurzivne funkcije
8. Primeri programa



1. Uvod

- Ponavljanje: pojam funkcije
- Ponavljanje: dekompozicija programskog koda

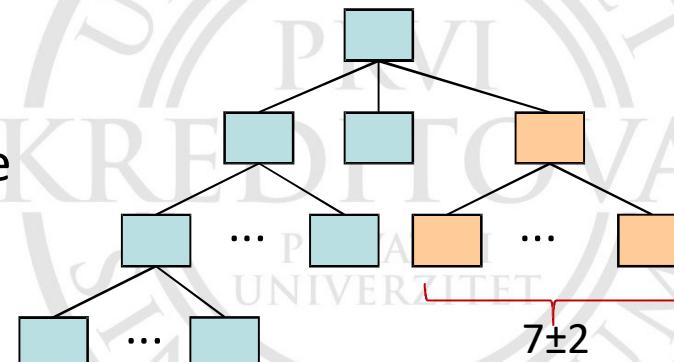


Ponavljanje: pojam funkcije

- Korisničke **funkcije** u proceduralnim jezicima su *imenovane grupe naredbi* koje izvršavaju određeni zadatak
- Funkcije omogućavaju
 - **višestruku upotrebu** dela programskog koda i time skraćenje dužine programa
 - **bolju organizaciju** koda programa njegovom podelom na manje celine, koje je lakše razumeti
- Funkcije imaju listu parametara koje koriste za izvršavanje određenog zadatka, npr. **strcpy(s1, s2)**
- Funkcije mogu (ali ne moraju) da kao rezultat vrate vrednost određenog tipa

Ponavljanje: dekompozicija programskog koda

- Dekompozicija programskog koda u manje celine neophodna je prilikom razvoja kvalitetnog softvera. Obiman programski kod podeli se na male, razumljive celine
- Višestruka dekompozicija proizvodi hijerarhiju manjih programskih celina
 - poželjno je na svakom nivou hijerarhije izvršiti podelu na ograničen broj manjih celina, jer celina koja se sastoji od velikog broja manjih delova sama postaje nerazumljiva
 - preporuka za *maksimalni* broj grananja u hijerahiji dekompozicije programa može biti tzv. "Milerov magični broj 7 ± 2 " [9]



2. Funkcije u jeziku C++

- Definisanje funkcije u jeziku C++
- Povratak iz funkcije
- Završetak programa iz funkcije
- Argumenti funkcije



Definisanje funkcije u jeziku C++

- Program u jeziku C++ predstavlja skup naredbi organizovanih u funkcije, koje imaju opšti oblik

```
tip_vrednosti naziv_funkcije (lista_parametara) {  
    // telo funkcije  
}
```

- **tip vrednosti** je tip podataka koje funkcija vraća kao rezultat i može da bude bilo koji tip osim polja. Funkcija ne mora da vraća rezultat i tada se deklariše kao **void** - funkcija bez tipa
- **naziv** funkcije je standardni identifikator
- **lista parametara** je niz identifikatora s oznakom *tipa* ispred, odvojenih zarezima. Funkcija ne mora da ima parametre, kao npr. **main()**
Parametri funkcije su promenljive koje preuzimaju vrednosti konkretnih argumenata iz poziva funkcije, *isključivo po poziciji*

Primer

- **Telo funkcije** su naredbe, a ako funkcija vraća neku vrednost, završava naredbom **return**
 - izvršavanje programa se nastavlja od prve sledeće naredbe iza poziva funkcije
- **Prototip funkcije** je deklaracija funkcije pre njene definicije, koja daje prevodiocu informaciju o *tipu rezultata, nazivu i parametrima funkcije*

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

// Prototip funkcije
void mojaFunkcija();

int main () {
    cout << "U funkciji main" << endl;
    mojaFunkcija(); // poziv funkcije
    cout << "Ponovo u main" << endl;
    return 0;
}

void mojaFunkcija(){
    cout << "U mojaFunkcija" << endl;
}
```

Rezultat:

U funkciji main
U mojaFunkcija
Ponovo u main

Povratak iz funkcije

- Funkcija koja ne vraća vrednost može da završi poslednjom naredbom, ali se obično eksplicitno završava naredbom `return`, s vrednošću koja se vraća ili bez vrednosti

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

void mojaFunkcija(); // prototip funkcije

int main () {
    cout << "Pre poziva" << endl;
    mojaFunkcija(); // poziv funkcije
    cout << "Posle poziva" << endl;
    return 0
}

void mojaFunkcija() {
    cout << "U funkciji" << endl;
    return;
    cout << "Ova naredba se neće izvršiti";
}
```

Rezultat:

Pre poziva
U funkciji
Posle poziva

povratak iz funkcije koja vraća celobrojni rezultat

povratak iz funkcije koja ne vraća rezultat

Završetak programa iz funkcije

- Program može završiti i u funkciji koja nije `main()`, za što se koristi posebna funkcija `exit()` iz biblioteke `cstdlib`

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

void mojaFunkcija() // prototip funkcije

int main () {
    mojaFunkcija(); // poziv funkcije
}

void mojaFunkcija() {
    cout << "Program završava pomoću funkcije exit" << endl;
    exit(0);
    cout << "Ova naredba se neće izvršiti";
}
```

Rezultat:

Program završava pomoću funkcije exit

završetak programa u funkciji `mojaFunkcija`, koja vraća operativnom sistemu kod 0 - uspešan završetak (konstante iz `cstdlib`: `EXIT_SUCCESS` i `EXIT_FAILURE`)

Argumenti funkcije

- Promenljive koje preuzimaju aktuelne vrednosti iz poziva funkcije su *parametri*, a same vrednosti *argumenti* funkcije:

```
// Program koji ilustruje pozivanje funkcija
#include <iostream>
using namespace std;
```

```
// Prototip funkcije
void kutija(int duzina, int sirina, int visina);
```

```
int main () {
    kutija (2, 3, 4); // poziv funkcije
    return 0;
}
```

```
// Definicija funkcije
void kutija(int duzina, int sirina, int visina) {
    cout << "Zapremina kutije je " << duzina * sirina * visina << endl;
}
```

*Rezultat (2*3*4):*
Zapremina kutije je 24

aktuelti argumenti

parametri funkcije

3. Oblast važenja promenljivih

- Životni vek i oblast važenja promenljivih
- Lokalne promenljive
- Globalne promenljive
- Statičke promenljive



Životni vek i oblast važenja promenljivih

- Promenljive programa imaju konačan životni vek, koji počinje njihovim deklarisanjem, a završava na više načina, npr. povratkom iz funkcije ili terminiranjem programa
- Prema životnom veku, razlikuju se *automatske*, *statičke* i *dinamičke* promenljive
 - *automatske*: lokalne u nekom kontekstu, postoje samo u steku poziva ili u registrima procesora i automatski se uklanjuju
- Promenljive su dostupne/važe u delu programa koji zavisi od mesta njihovog deklarisanja i mogu biti *lokalne* ili *globalne*:
 - **lokalno dostupne** (*local scope*) su promenljive definisane u okviru bloka naredbi i kao parametri funkcija
 - **globalno dostupne** (*global scope*) su promenljive definisane izvan svih blokova naredbi i klasa

Lokalne promenljive

- Lokalno dostupne (*local scope*) promenljive definisane su unutar nekog bloka naredbi i kao parametri funkcija
- Promenljive nisu vidljive niti dostupne izvan bloka, a sve promene su lokalne za taj blok naredbi (najčešće funkciju)
- Promenljive postoje samo od njene deklaracije do završetka izvršavanja bloka naredbi
- Nakon izvršavanja bloka naredbi ili funkcije, sve lokalne promenljive se uništavaju
- Višestrukim pozivanjem funkcije, lokalne promenljive se svaki put ponovo kreiraju, koriste i uništavaju po okončanju rada

Primer: Lokalni kontekst imena promenljivih

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

void mojaFunkcija(); // prototip funkcije

int main () {
    int promenljiva = 10;           ← promenljiva je vidljiva u funkciji main()
    cout << "Vrednost promenljive u funkciji main = " << promenljiva << endl;
    mojaFunkcija();   // poziv funkcije
    cout << "Vrednost promenljive u funkciji main " <<
        "nakon poziva funkcije = " << promenljiva << endl;
    return 0;
}

// Definicija funkcije
void mojaFunkcija() {
    int promenljiva = 88;          ← promenljiva je vidljiva u funkciji mojaFunkcija()
    cout << "Vrednost promenljive u funkciji = " << promenljiva << endl;
}

Rezultat:
Vrednost promenljive u funkciji main = 10
Vrednost promenljive u funkciji = 88
Vrednost promenljive u funkciji main nakon poziva funkcije = 10
```

Primer: Vrednost lokalne promenljive u funkciji

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

void mojaFunkcija(); // prototip funkcije

int main () {

    for (int i=0; i<3; i++)
        mojaFunkcija(); // poziv funkcije

    return 0;
}

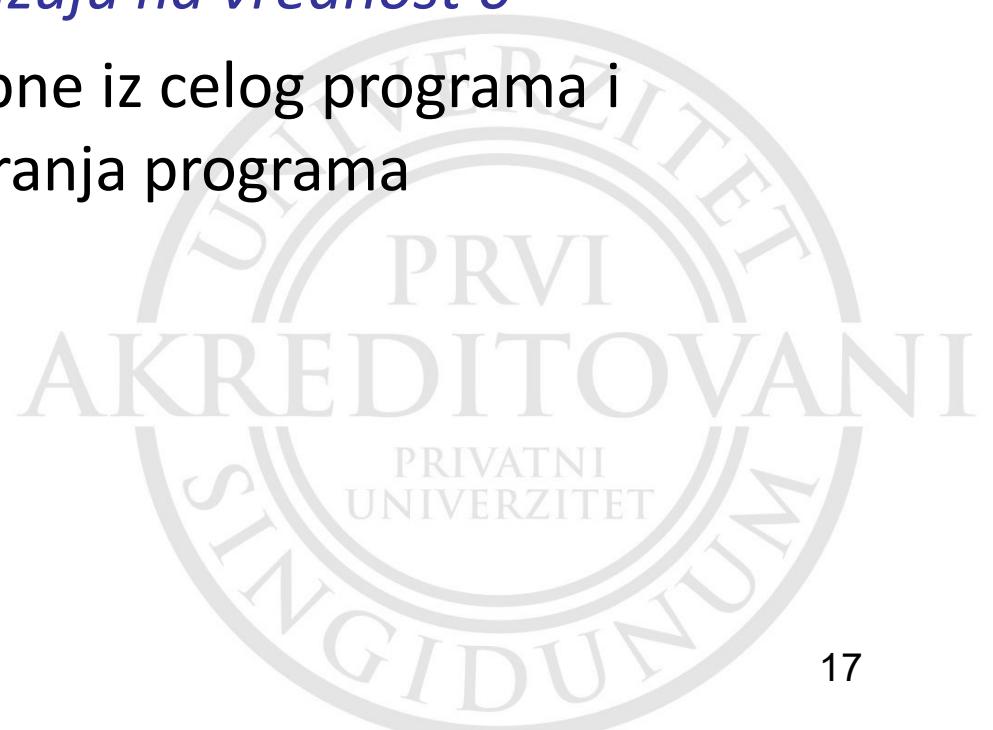
// Definicija funkcije
void mojaFunkcija() {
    // promenljiva se inicijalizuje prilikom svakog poziva funkcije
    int promenljiva = 88;
    cout << "Vrednost promenljive u funkciji = " << promenljiva << endl;
}
```

Rezultat:

Vrednost promenljive u funkciji = 88
Vrednost promenljive u funkciji = 88
Vrednost promenljive u funkciji = 88

Globalne promenljive

- Globalno dostupne (*global scope/global namespace/file scope*) su promenljive definisane izvan svih blokova naredbi i klasa, npr. na početku fajla
- Ako se prilikom definisanja promenljive ne navede njihova vrednost, *automatski se inicializuju na vrednost 0*
- Globalne promenljive su dostupne iz celog programa i zadržavaju vrednost do terminiranja programa



Primer: Gobalna promenljiva vidljiva u celom programu

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

void mojaFunkcija1(); // prototip funkcije
void mojaFunkcija2(); // prototip funkcije

int brojac; // globalna promenljiva brojac ← promenljiva je vidljiva celom programu

int main () {
    for (int i=0; i<10; i++) {
        brojac = i * 2; // vrednost globalne promenljive brojac
        mojaFunkcija1(); // poziv funkcije 1 za svaku vrednost brojaca
    }
    return 0;
}

void mojaFunkcija1() {
    // pristup globalnoj promenljivoj brojac
    cout << "Brojac = " << brojac << endl;
    mojaFunkcija2(); // poziv funkcije 2
}

void mojaFunkcija2() {
    int brojac; // lokalna promenljiva brojac
    for (brojac=0; brojac<3; brojac++)
        cout << ".";
}
```

Rezultat:

```
Brojac = 0
...Brojac = 2
...Brojac = 4
...Brojac = 6
...Brojac = 8
...Brojac = 10
...Brojac = 12
...Brojac = 14
...Brojac = 16
...Brojac = 18
...
```

promenljiva je vidljiva samo u funkciji mojaFunkcija2()

Alokacija globalnih promenljivih

- Globalne promenljive se kreiraju u posebnoj oblasti memorije i dostupne su iz celog programa, sve dok ne terminira
- Upotrebu globalnih promenljivih u programu treba ograničiti na neophodnu meru, jer
 - zauzimaju memoriju celo vreme izvršavanja programa
 - funkcije koje koriste globalne promenljive postaju *zavisne*, tako da je smanjena njihova višetruka upotrebljivost
 - zavisnost od globalnih promenljivih i promena njihove vrednosti prouzrokuju neželjene interakcije i pojavu greški na različitim mestima u programu koje se teško otkrivaju
- Pamćenje vrednosti između poziva neke funkcije može se realizovati korišćenjem *statičkih promenljivih*

Statičke promenljive

- Statičke promenljive su *lokalne* za funkciju ili blok naredbi, ali se inicijalizuju *samo jednom*, prilikom prvog izvršavanja funkcije ili bloka naredbi. Definišu se pomoću modifikatora `static tip naziv;`
- Statičke promenljive su dostupne iz bloka u kome su kreirane, a njihova vrednost se pamti do kraja izvršavanja programa
- Napomena:
 - modifikator `static` može se primeniti i na *globalne* promenljive, za ograničavanje njihove vidljivosti u velikim programima
Iako *globalne*, tako definisane promenljive su dostupne samo *u okviru fajla* u kome su definisane, dok ostale funkcije programa ne mogu da promene njihovu vrednost

Primer: Upotreba statičke promenljive za brojanje poziva funkcije

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

void zapamti(); // prototip funkcije

int main () {
    for (int i=0; i<3; i++)
        zapamti(); // poziv funkcije
    return 0;
}

void zapamti() {
    // promenljiva se inicijalizuje samo prilikom prvog poziva funkcije
    static int brojac = 0;
    cout << "Ovo je " << ++brojac << ". poziv funkcije" << endl;
}
```

Rezultat:

Ovo je 1. poziv funkcije
Ovo je 2. poziv funkcije
Ovo je 3. poziv funkcije

4. Prostori imena (imenici)

- Deklarisanje prostora imena (imenika)
- Pristup imenima
- Naredba *using*



Deklarisanje prostora imena (imenika)

- Samo jedan objekt s određenim nazivom može da postoji u nekoj oblasti važenja promenljivih
- Lokalna imena su pod kontrolom programera, ali se može pojaviti konflikt s imenima promenljivih koje nisu lokalne, npr. s imenima u okviru biblioteka funkcija
- Prostori imena (imenici) omogućavaju grupisanje imenovanih objekata i definisanje užih oblasti važenja njihovih naziva
- Prostori imena (*namespaces*) takođe imaju naziv i deklarišu se naredbom:

```
namespace naziv {  
    imenovani_entiteti  
}
```

Pristup imenima

- Sva imena definisana u okviru nekog prostora imena imaju njegovu oblast važnosti. Prostor imena se uvodi naredbom *using namespace*, koja uvozi sva imena definisana u imeniku, tako da im se može direktno pristupati po nazivu
- Imenima iz imenika može se pristupiti i bez ove naredbe korišćenjem operatora opsega važnosti *:: (scope resolution)*

```
#include <iostream>
namespace imenik {
    int vrednost = 0;
}

int main () {
    std::cout << "Unesite ceo broj: ";
    std::cin >> imenik::vrednost;
    std::cout << std::endl << "Uneli ste: " << imenik::vrednost << std::endl;
    return 0;
}
```

Rezultat:

Unesite ceo broj: 2

Uneli ste: 2

Naredba *using*

- Naredbom *using namespace* uvoze se kompletni imenici
- Naredbom *using* mogu se uvoziti i pojedinačna imena, npr.

```
#include <iostream>
using namespace std;
namespace prvi {
    int x = 1;
};
namespace drugi {
    double x = 1.41;
};

int main () {
    using namespace drugi
    cout << x << endl;
    using prvi::x;
    cout << x << endl;
    using drugi::x;
    cout << x << endl;
    return 0;
}
```

Rezultat:

1.41
1
1.41

5. Prenos argumenata funkcije

1. Prenos argumenata
2. Prenos struktura kao argumenata
3. Reference (adrese) u funkcijama
4. Samostalne reference
5. Pokazivači i reference
6. Referenca kao tip funkcije
7. Parametri funkcije i modifikator const



5.1 Prenos argumenata

- Prenos argumenata funkciji može se izvršiti
 - po vrednosti (*pass by value*), način koji se podrazumeva ili
 - po adresi (*pass by reference*)
- Prenos po vrednosti znači da se *kopira vrednost* argumenta u parametar funkcije, tako da promene vrednosti parametra unutar funkcije *ne utiču* na vrednost argumenta koji je funkciji prosleđen
- Prenos po adresi vrši se tako da se *kopira adresa* aktuelnog argumenta, a ne vrednost i sve promene parametra unutar funkcije *menjaju* i sam argument
Prenos po adresi realizuje se tako što se kao argument funkcije koristi *pokazivač*

5.2 Prenos struktura kao argumenata

- Argumenti funkcija mogu biti *pokazivači* i *polja*. Za prenos pokazivača, parametre funkcije treba definisati kao *pokazivače*, npr.

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

void mojaFunkcija(int *p); // prototip funkcije

int main () {
    int i = 0;
    int *p; p = &i      // p je pokazivač na i
    // prosleđivanje pokazivača kao argumenta funkcije
    mojaFunkcija(p);
    cout << i << endl; // nova vrednost i je 100
    return 0;
}

void mojaFunkcija(int *p){
    // promenljivoj na koju pokazuje p dodeljuje se vrednost 100
    *p = 100;
}
```

parametar funkcije definisan kao pokazivač;
vrednost se dodeljuje promenljivoj na koju pokazuje

Rezultat:
100

Prenos polja kao argumenata (1)

- Ako je parametar funkcije *polje*, kao argument se navodi polje koje se prenosi kao *pokazivač* na prvi element, npr.

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

void prikaziPolje(int brojevi[10]); // prototip funkcije

int main () {
    int polje[10], i;
    // Inicijalizacija polja
    for (int i=0; i<10; i++)
        polje[i] = i;
    // Prosleđivanje polja kao argumenta: pokazuje na prvi element
    prikaziPolje(polje);
    return 0;
}

void prikaziPolje(int brojevi[10]){
    for (int i=0; i<10; i++)
        cout << brojevi[i] << " ";
    cout << endl;
}
```

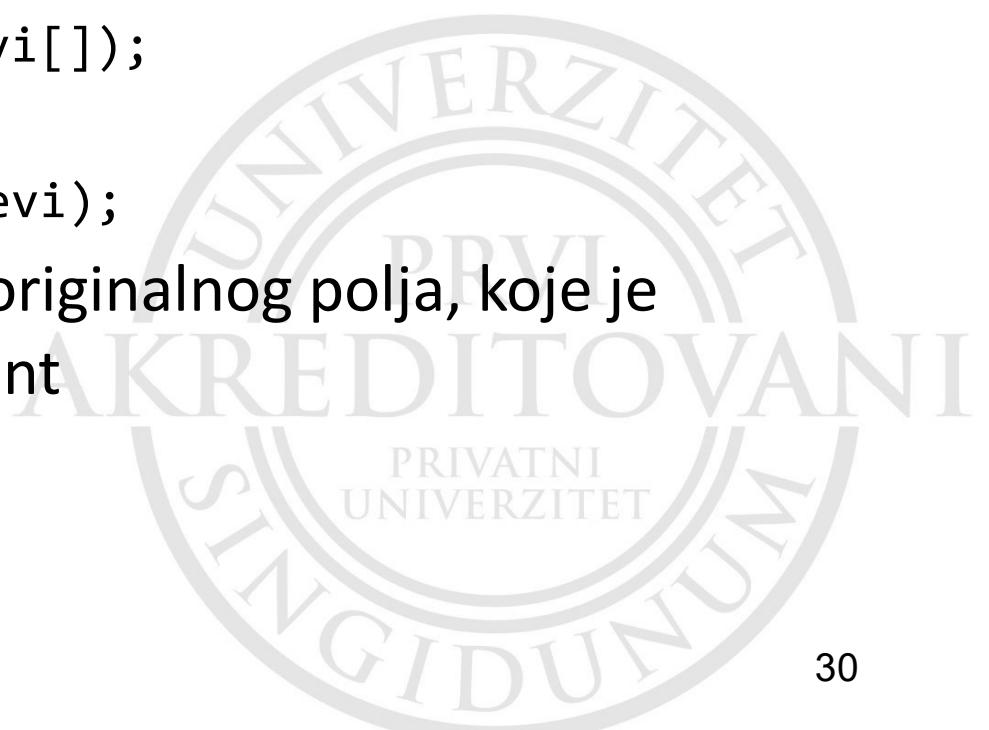
parametar funkcije definisan kao polje

Rezultat:

0 1 2 3 4 5 6 7 8 9

Prenos polja kao argumenata (2)

- Za prenos polja kao argumenta funkcije, prametar funkcije se može definisati na više načina, kao
 - polje istog tipa i dimenzija kao aktuelni argument, npr.
`void prikaziPolje(int brojevi[10]);`
 - polje bez dimenzija , npr.
`void prikaziPolje(int brojevi[]);`
 - pokazivač na polje , npr.
`void prikaziPolje(int *brojevi);`
- Funkcija uvek *menja* vrednosti originalnog polja, koje je preneseno kao aktuelni argument



Primer: Promena vrednosti argumenta funkcije tipa polja

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

void prikaziPolje(int brojevi[10]); // prototip funkcije
void kub(int *p, int n); // prototip funkcije

int main () {
    int polje[10], i;
    for (int i=0; i<10; i++) polje[i] = i+1; // inicijalizacija polja
    cout << "Originalni niz: ";
    prikaziPolje(polje);

    kub(polje, 10); // adresa polja i dimenzija kao argumenati
    cout << "Izmenjeni niz: ";
    prikaziPolje(polje);

    return 0;
}

void prikaziPolje(int brojevi[10]){ // definicija funkcije
    for (int i=0; i<10; i++)
        cout << brojevi[i] << " ";
    cout << endl;
}

void kub(int *p, int n){ // definicija funkcije
    while (n) {
        *p = *p * *p * *p;
        n--; p++;
    }
}
```

- prvi parametar deklarisan kao pokazivač na int (prvi element polja)
- drugi parametar je int (dimenzija polja)

- parametar deklarisan kao pokazivač na int (prvi element polja)
- drugi parametar je dimenzija polja

Rezultat:

```
Originalni niz: 1 2 3 4 5 6 7 8 9 10
Izmenjeni niz: 1 8 27 64 125 216 343 512 729 1000
```

5.3 Reference (adrese) u funkcijama

- Podrazumevajući način prenosa argumenata u jeziku C++ je *po vrednosti*.
Eksplisitim korišćenjem pokazivača moguće je ostvariti prenos argumenata *po adresi* (referenci), tako da se prenose adrese aktuelnih argumenata umesto njihovih vrednosti
- Eksplisitna primena pokazivača nije praktična, jer se mogu dogoditi greške, ako se u pozivu umesto adresa prenesu vrednosti argumenata
- Bolje rešenje je ako se prevodiocu u definiciji funkcije pomoću operatora & navede da su parametri *reference*, a ne vrednosti promenljivih. Tada se adrese prenose automatski, poziv funkcije je uobičajen, a u kodu se ne koriste oznake *

Primer: Funkcija zamene vrednosti argumenata na dva načina

```
#include <iostream>
using namespace std;
// Prototip funkcije zamene argumenata
void swap(int *x, int *y)

int main() {
    int i=10, j=20;
    cout << "Pre:" << i << ", " << j << endl;
    // Funkciji swap prosledjuju se adrese arg.
    swap(&i, &j);
    cout << "Posle: << i << ", " << j << endl;
    return 0;
}

void swap(int *x, int *y) {
    int temp;
    temp = *x; // sacuva vrednost na adresi x
    *x = *y;   // smesti y u x
    *y = temp; // smesti x u y
}
```

Rezultat:
Pre:10, 20
Posle:20, 10

```
#include <iostream>
using namespace std;
// Prototip funkcije ciji su arg. reference
void swap(int &x, int &y)

int main() {
    int i=10, j=20;
    cout << "Pre:" << i << ", " << j << endl;
    // Funkciji swap prosledjuju se argumenti
    swap(i, j);
    cout << "Posle: << i << ", " << j << endl;
    return 0;
}

void swap(int &x, int &y) {
    int temp;
    temp = x; // sacuva vrednost na adresi x
    x = y;   // smesti y u x
    y = temp; // smesti x u y
}
```

5.4 Samostalne reference

- Adresa (referenca) može se koristiti kao alternativni način pristupa promenljivoj, jer je tokom celog životnog veka vezana za objekt na koji je inicijalizovana, npr.

```
long broj = 10;
```

```
long &ref_broj = broj; // referenca na promenljivu broj
```



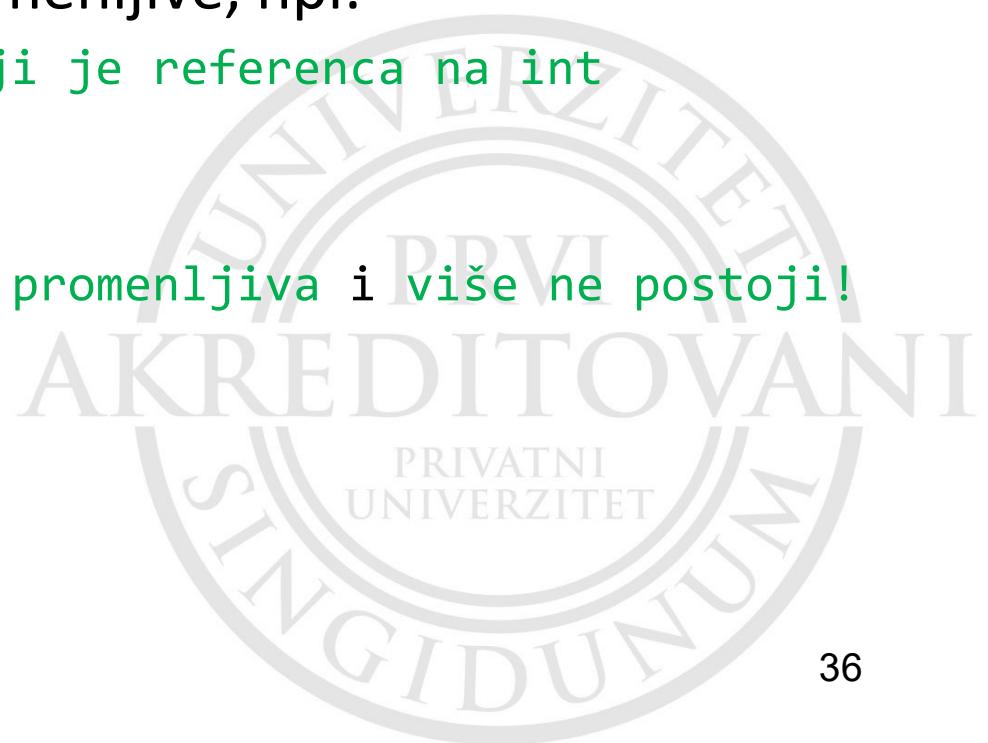
5.5 Pokazivači i reference

- Postoje razlike između pokazivača i referenci objekata:
 - pokazivač se može promeniti tako da pokazuje na drugi objekt (promenljivu), a referenca ne može
 - pokazivač može da poprими vrednost **NULL**, a referenca uvek pokazuje na određeni objekt
 - pristup objektu preko pokazivača vrši se posredno pomoću operatora *****, a pristup preko reference je direktn
 - moguće je kreiranje polja pokazivača, ali ne i polja referenci
 - za razliku od pokazivača, nema reference na referencu

5.6 Referenca kao tip funkcije

- Referenca može da bude *tip rezultata* funkcije, ali treba koristiti samo reference na objekte koji postoje i *nakon završetka* izvršavanja funkcije
- Ne treba koristiti lokalne promenljive i argumente funkcija, kao ni reference na lokalne promenljive, npr.

```
// Funkcija vraća rezultat koji je referencia na int
int &funkcija() {
    int i = 10;
    return i; //nakon završetka promenljiva i više ne postoji!
}
```



5.7 Parametri funkcije i modifikator const

- Parametar funkcije može se zaštiti od promene vrednosti korišćenjem modifikatora **const**
- Modifikator omogućava prevodiocu da proverava i upozori na pokušaj promene vrednosti argumenta



6. Prototipovi i preklapanje funkcija

- Prototip funkcije
- Preklapanje funkcija
- Podrazumevani argumenti funkcija



Prototip funkcije

- U jeziku C++ sve funkcije moraju da budu deklarisane pre prve upotrebe
- Prototip funkcije sadrži osnovne elemente definicije funkcije, koji su neophodni prevodiocu za njenu upotrebu
 - tip rezultata
 - broj argumenata
 - tip argumenata
- Npr. prototip funkcije koja očekuje pokazivač kao argument:

```
void funkcija(int *p);
```

```
int main() {
    int x = 10;
    funkcija(x); // greška, argument nije pokazivač!
}
```



Prototip funkcije

- Opšti oblik prototipa odgovara definiciji funkcije bez tela funkcije
- Prototipovi pomažu prevodiocu da
 - ustanovi koji je tip rezultata funkcije, odnosno koju vrstu koda generiše
 - pronađe neispravne konverzije između tipova argumenata u pozivima funkcije i definicijama parametara
 - proveri slaganje broja argumenata u pozivu i definiciji funkcije
- Datoteke zaglavlja (*header files*) sadrže prototipove svih funkcija koje se nalaze u određenoj biblioteci
- Funkcija `main()` je izuzetak: može da postoji samo jedna, bilo gde u programu i za nju ne postoji prototip, jer je ugrađena u jezik C++

Preklapanje funkcija

- U jeziku C++ je dozvoljeno da postoji više funkcija istog imena, ako imaju različit broj i tipove argumenata
- Ovakvo **preklapanje funkcija** (*function overloading*) omogućava upotrebu funkcija istog naziva za iste zadatke na različitim tipovima podataka (*polimorfizam*) npr.

```
int min(int a, int b);      // funkcija min za cele brojeve
char min(char a, char b);   // funkcija min za znakove
```
- Preklapanje postoji samo za funkcije kojima su različiti tipovi rezultata ili broj argumenata, a prevodilac bira odgovarajuću funkciju prema tipovima argumenata

Primer: Preklopljene funkcije

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

// Prototipovi preklopljenih funkcija
int min(int a, int b);      // funkcija min za cele brojeve
char min(char a, char b);   // funkcija min za znakove

int main () {
    cout << min(2, 3) << endl;
    cout << min('a', 'B') << endl;
    return 0;
}

int min(int a, int b) {
    int min;
    (a < b) ? min=a : min=b;
    return min;
}

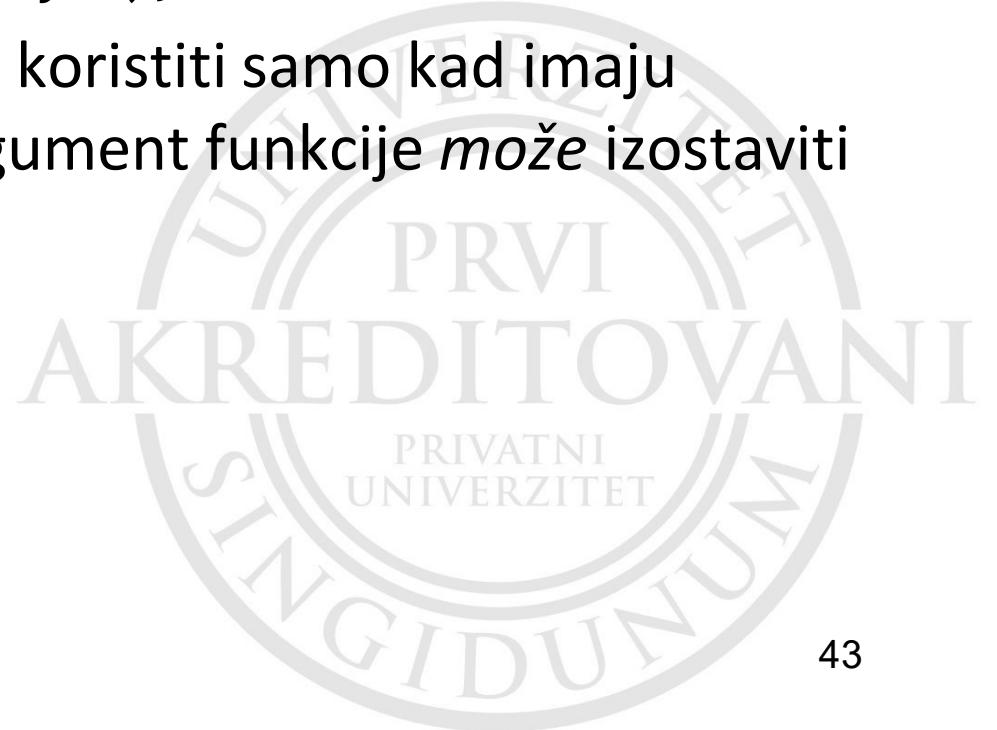
// Funkcija za znakove izjednačava mala i velika slova
char min(char a, char b) {
    char min;
    (tolower(a) < tolower(b)) ? min=a : min=b;
    return min;
}
```

Rezultat:

2
a

Podrazumevani argumenti funkcija

- Parametri funkcije mogu da imaju podrazumevanu (*default*) vrednost, koja se koristi kad se vrednost argumenta izostavi u pozivu funkcije
- Sintaksa je kao kod inicijalizacije promenljive, npr.
`void mojaFunkcija(int x=0, int y=0);`
- Podrazumevane vrednosti teba koristiti samo kad imaju smisla, odnosno kad se neki argument funkcije *može* izostaviti



7. Rekurzivne funkcije

- Pojam rekurzije
- Primer



Pojam rekurzije

- Rekurzija omogućava definisanje nekog pojma pomoću njega samog
- Rekurzija u programiranju je poseban način ponavljanja grupa naredbi pomoću funkcija koje pozivaju same sebe
- **Rekurzivne funkcije** u jeziku C++ su funkcije koje pozivaju same sebe u telu funkcije
 - prilikom svakog aktiviranja rekurzivne funkcije, aktuelni parametri funkcije se čuvaju u *steku*, koji ima ograničenu veličinu. Da se izbegne iscrpljivanje steka, definiše se najveća dozvoljena dubina rekurzije
 - rekurzivno definisana funkcija treba u telu funkcije da obezbedi uslov okončanja rekurzije (slično uslovu petlje)
 - rekurzija je *neefiksan* način ponavljanja; rekurzivni algoritmi se mogu realizovati iterativno, pomoću petlji i strukture stek

Primer: Ispis teksta unazad

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

void ispisStringaUnazad(char*);

int main () {
    char str[] = "Ana voli Milovana";
    ispisStringaUnazad(str);
    return 0;
}

void ispisStringaUnazad(char *p) {
    if (*p)
        ispisStringaUnazad(p+1) // rekurzivni poziv za sledeći znak
    else
        return; // povratak bez ispisa (kraja stringa)
    cout << *p; // ispis tekućeg znaka nakon povratka iz rekurzije
}
```

Rezultat:
anavoliM ilov anA

8. Primeri programa

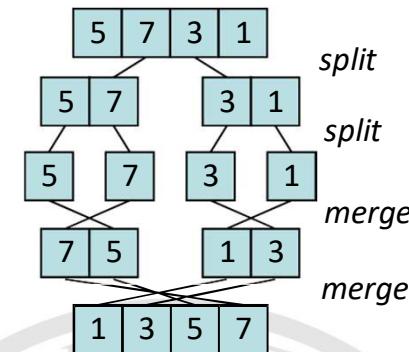
1. Sortiranje polja: Merge sort
2. Najkraći put u grafu: Dijkstrin algoritam



8.1 Sortiranje polja: Merge sort

John von Neumann, 1945

- Rekursivni algoritam sortiranja niza (polja) asimptotske složenosti $O(n \log n)$ ima opšti oblik:
 1. **Podeli** se niz na dva podniza (*split*)
 2. **Sortira** se svaki podniz
 3. **Spoje** se dva sortirana podniza u jedan sortirani niz (*merge*)
- Podela na dva podniza vrši se na najjednostavniji način, sekvencijalnom podelom *na sredini* niza
- Spajanje sortiranih nizova u jedan vrši se poređenjem prvih elemenata oba podniza i premeštanjem manjeg/većeg elementa u izlazni niz



Program (1/2)

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

// Spajanje sortiranih podnizova (merge)
void merge(int array[], int temp[], int left, int right) {
    int middleIndex = (left + right)/2;
    int leftIndex = left;
    int rightIndex = middleIndex + 1;
    int tempIndex = left;
    while (leftIndex <= middleIndex && rightIndex <= right) {
        if (array[leftIndex] >= array[rightIndex])
            temp[tempIndex] = array[leftIndex++];
        else
            temp[tempIndex] = array[rightIndex++];
        tempIndex++;
    }
    while (leftIndex <= middleIndex) {
        temp[tempIndex] = array[leftIndex++];
        tempIndex++;
    }
    while (rightIndex <= right) {
        temp[tempIndex] = array[rightIndex++];
        tempIndex++;
    }
}
```



Program (2/2)

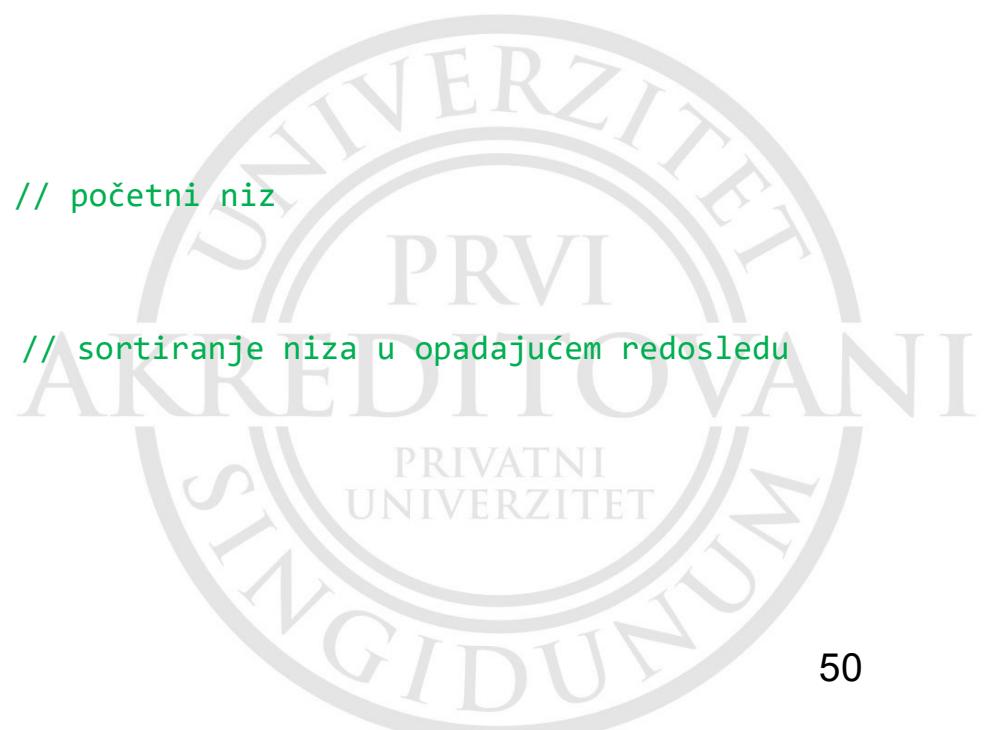
```
// Sortiranje spajanjem (Merge sort) u opadajućem redosledu
void mergeSort(int array[], int temp[], int left, int right) {
    if (left == right) return;
    int middleIndex = (left + right)/2;
    mergeSort(array, temp, left, middleIndex);
    mergeSort(array, temp, middleIndex + 1, right);
    merge(array, temp, left, right);

    for (int i = left; i <= right; i++) {
        array[i] = temp[i];
    }
}

int main() {
    int polje[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; // početni niz
    int brojElemenata = 10;
    int temp[10];

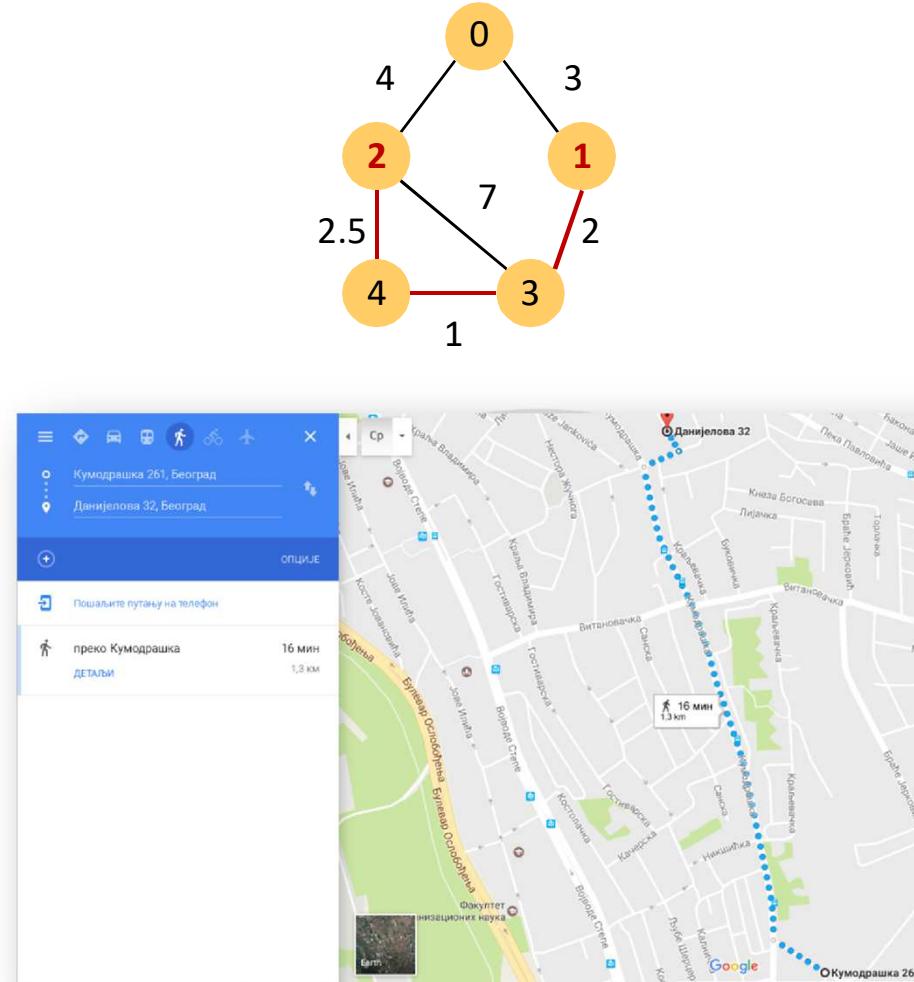
    mergeSort(polje, temp, 0, brojElemenata-1);           // sortiranje niza u opadajućem redosledu

    for (int i=0; i<brojElemenata; i++)
        cout << polje [i] << " ";
    cout << endl;
    return 0;
}
```



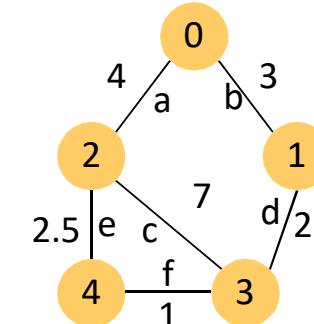
8.2 Najkraći put u grafu: Dijkstrin algoritam (1/4)

- Primer upotrebe matrica u rešavanju grafovskih problema
 - graf $G(v,e)$ je skup čvorova v (*vertex, node*) povezanih lukovima e (*edge*)
 - osim oznaka, *lukovi* grafa mogu imati i različita kvantitativna obeležja, koja predstavljaju npr. cenu, dužinu i sl.
 - pretraživanjem je moguće pronaći *najkraći put* između dva čvora grafa
- Ako čvorovi grafa predstavljaju naselja, a lukovi direktne puteve između njih, *najkraći put u grafu* odgovara najkraćem putu između dva naselja (lokacije)
 - slično www.google.com/maps



Najkraći put u grafu: Dijkstrin algoritam (2/4)

- Grafovi se mogu predstaviti matricama
 - matrica susedstva (*adjacency*)
 $A = [a_{ij}]$ pokazuje da li su dva čvora povezana lukom. Indeksi i i j su indeksi čvorova. a elementi matrice a_{ij} mogu da označavaju npr. da li su *dva čvora* povezana (0 ili 1), *broj lukova* ili *dužinu luka* ij
 - matrica incidencije $B = [b_{ij}]$
 pokazuje da li je neki čvor povezan s nekim lukom. Indeks i je čvor, j je oznaka luka, a element matrice $b_{ij} = 0$ ili 1 označava da li je čvor i povezan (incidentan) s lukom j



	0	1	2	3	4
0	0	3	4	-	-
1	3	0	-	2	-
2	4	-	0	7	2.5
3	-	2	7	0	1
4	-	2.5	1	0	0

	a	b	c	d	e	f
0	1	1	0	0	0	0
1	0	1	0	1	0	0
2	1	0	1	0	1	0
3	0	0	1	1	0	1
4	0	0	0	0	1	1

Najkraći put u grafu: Dijkstrin algoritam (3/4)

- Jedan od najpoznatijih algoritama za pronalaženje najkraćeg puta između dva čvora u grafu je *Dijkstrin* algoritam
 - Osnovna verzija Dijstrinog algoritma predstavlja varijantu pretraživanja u dubinu (*breadth-first-search*), jer posećuje čvorove po redosledu njihovog pronalaženja. Za pamćenje njihovog redosleda koristi se struktura reda čekanja (*queue*), u koju se novi elementi dodaju na kraj reda, a uzimaju s početka reda
 - Prilikom pretraživanja, poseta svakom čvoru se označava, kako bi se kasnije posećenost mogla proveriti i pretraživanje nastavilo obilaskom ostalih povezanih, još neposećenih čvorova

Najkraći put u grafu: Dijkstrin algoritam - tri verzije (4/4)

a) Pojednostavljena verzija *Dijkstrinog* algoritma pretpostavlja jednake težine svih lukova (=1):

1. Početnom čvoru s dodeli se rastojanje 0 i on se dodaje u red čekanja
Ostalim čvorovima se dodeli beskonačno rastojanje od početnog čvora
2. Sve dok red čekanja nije prazan:
 1. Uklanja se iz reda čekanja prvi čvor i dodeli mu se rastojanje d
 2. Pronađu se svi lukovi povezani s ovim čvorom
 3. Za svaki povezani čvor s beskonačnim rastojanjem,
rastojanje se zameni s $d+1$, a čvor se dodaje na kraj reda čekanja

b) Osnovna verzija algoritma ne pretpostavlja jednake težine/dužine lukova, čvorove označava dužinom najkraćeg puta (od početnog čvora) i pronalazi najkraće puteve od početnog čvora *do svih ostalih* čvorova grafa

c) Unapređena verzija *Dijkstinog* algoritma koristi strukturu *prioritetnog reda* čekanja umesto običnog radi značajnog unapređenja performansi

Program (1/3)

```
// Najkraći pute u grafu prema Dijkstrinom algoritmu

#include <iostream>
using namespace std;

// Broj čvorova grafa
#define V 5

// Funkcija koja pronalazi najbliži čvor
// u skupu čvorova koji još nisu deo najkraćeg puta
int minDistance(int dist[], bool sptSet[]) {

    // Minimalno rastojanje
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// Prikaz liste rastojanja do svih čvorova
int printSolution(int dist[], int src, int n) {
    cout << "Čvor Rastojanje od cvora " << src << endl;
    for (int i = 0; i < V; i++)
        if (i != src)
            cout << i << " " << dist[i] << endl;
    return 0;
}
```



Program (2/3)

```
// Dijkstrin algoritam za pronalaženje najkraćih putevac od nekog čvora do svih ostalih čvorova grafa
void dijkstra(int graph[V][V], int src) {

    int dist[V];      // rezultat: dist[i] su najkraća rastojanja od čvora src do i
    bool sptSet[V];   // sptSet[i] je true ako je čvor i uključen u najkraći put
                      // ili je najkraći put od src do i pronađen

    // Inicijalizacija: distanci na INFINITE i sptSet[] na false
    for (int i = 0; i < V; i++)
        dist[i] = INT_MAX, sptSet[i] = false;

    dist[src] = 0;    // rastojanje od čvora do njega samog je 0

    // Najkraći put do svih ostalih čvorova
    for (int count = 0; count < V-1; count++) {

        // Pronalaženje minimalnog rastojanja od početnog čvora do čvorova
        // iz skupa još neobrađenih čvorova (u prvoj iteraciji čvor src)
        int u = minDistance(dist, sptSet);

        // Označavanje posećenog čvora
        sptSet[u] = true;

        // Ažuriranje rastojanja čvorova koji su susedi posećenog čvora
        for (int v = 0; v < V; v++)
            // Rastojanje se ažurira ako (1) čvor v nije u skupu uključenih,
            // (2) postoji luk od u do v i (3) ukupna dužina od src do v je manja od dist[v]
            if (!sptSet[v] && graph[u][v] && dist[u] != INT_MAX && dist[u]+graph[u][v] < dist[v])
                dist[v] = dist[u] + graph[u][v];
    }
    // Prikaz liste rastojanja
    printSolution(dist, src, V);
}
```



Program (3/3)

```
// Testni program
int main() {

    cout << "Program pronalazi najkraci put između dva cvora u grafu" << endl;
    cout << "koji je definisan matricom susedstva (adjacency matrix):" << endl;
    cout << "          (0)" << endl;
    cout << "      4 | 3 | 0 3 4 - - |" << endl;
    cout << "      | | 3 0 - 2 - |" << endl;
    cout << " (2)   (1) | 4 - 0 7 3 |" << endl;
    cout << "      | | 7 | - 2 7 0 1 |" << endl;
    cout << "      3 | | 2 | - - 3 1 0 |" << endl;
    cout << "      (4)--(3)" << endl;
    cout << "          1" << endl;

    // Matrica susedstva
    int graph[V][V] = {{0, 3, 4, 0, 0},
                        {3, 0, 0, 2, 0},
                        {4, 0, 0, 7, 3},
                        {0, 2, 7, 0, 1},
                        {0, 3, 1, 0, 0}
                      };
    dijkstra(graph, 2);

    system("pause");

    return 0;
}
```

Program pronalazi najkraci put između dva cvora u grafu
 koji je definisan matricom susedstva (adjacency matrix):

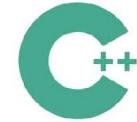
	(0)								
4		3		0	3	4	-	-	
				3	0	-	2	-	
(2)		(1)		4	-	0	7	3	
				-	2	7	0	1	
3		2		-	-	3	1	0	
(4)--(3)									
1									

Cvor Rastojanje od cvora 2

0	4
1	6
3	7
4	3

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learnCPP.com/>
 - <http://www.programming-algorithms.net/article/39650/Merge-sort>
<http://www.geeksforgeeks.org/greedy-algorithms-set-6-dijkstras-shortest-path-algorithm/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 04

Objektno orijentisano programiranje u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod (podsetnik)
2. Klase i objekti u jeziku C++
3. Konstruktori i destrukturatori
4. Upotreba objekata
5. Klasa string
6. Nasleđivanje i izvedene klase u jeziku C++



1. Uvod (podsetnik)

- Objektno orijentisani razvoj softvera
- Pojam objekta i klase

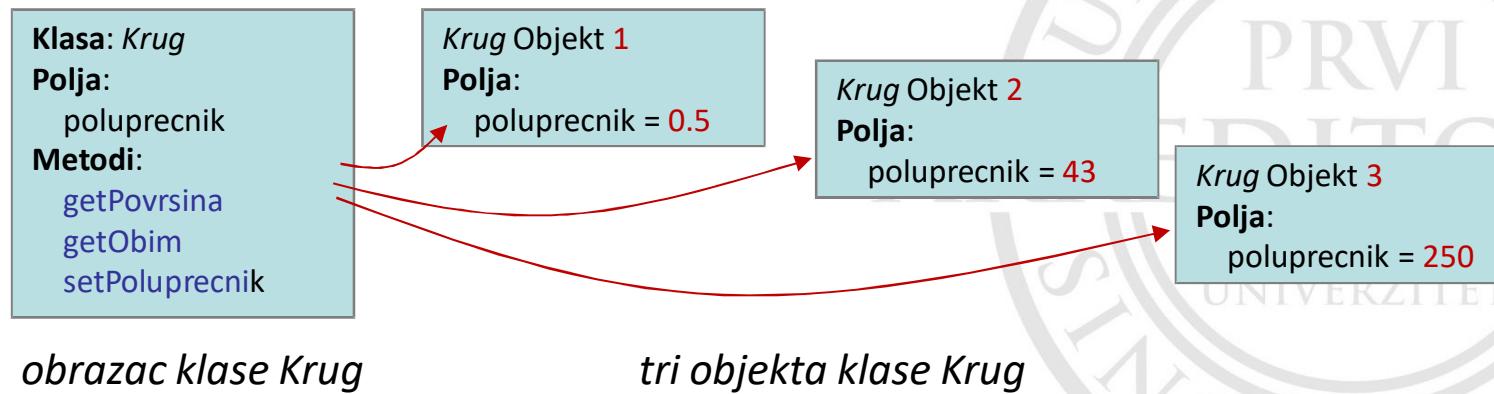


Objektno orijentisani razvoj softvera

- Objektno orijentisani pristup rešava mnoge probleme koji su svojstveni proceduralnim programiranju, gde su podaci i operacije su međusobno *razdvojeni*, tako da je neophodno slanje podataka metodima
- Objektno orijentisano **programiranje** smešta podatke i operacije koje se na njih odnose *zajedno*, u objektu
 - program se može posmatrati kao kolekcija *objekata* koji međusobno sarađuju
- Korišćenje objekata poboljšava višestruku upotrebljivost softvera i program čini lakšim za razvoj i održavanje

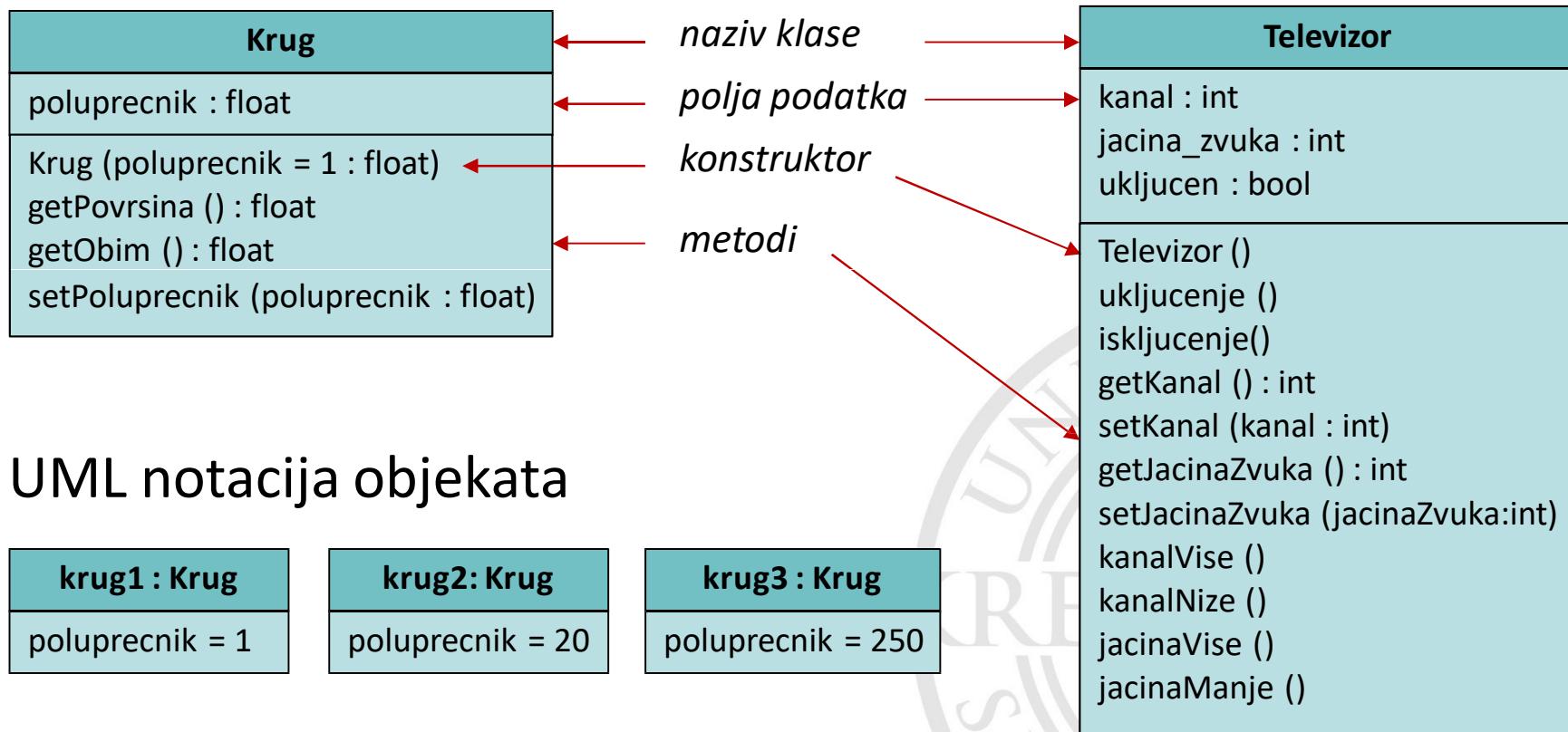
Pojam objekta i klase

- Klasa predstavlja opis objekata iste vrste, koji imaju slična svojstva: analogija *recepta* za kolač (klasa) i *kolača* (objekt)
 - korisnički definisana klasa koristi promenljive kao polja za smeštanje podataka i definiše metode za izvršavanje akcija
 - predstavlja obrazac (*template*), nacrt (*blueprint*) ili ugovor (*contract*)
- Objekt je primerak ili instanca (*instance*) klase
- Kreiranje instance klase se naziva instancijacija (*instantiation*)

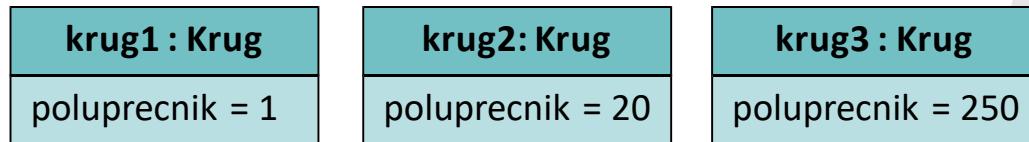


Prikaz klasa u jeziku UML

- UML dijagrami klase



- UML notacija objekata



2. Klase i objekti u jeziku C++

1. Deklarisanje klase
2. Enkapsuliranje klase
3. Definisanje funkcija članova klase
4. Pristup članovima klase
5. Ugrađene (inline) funkcije



2.1 Deklarisanje klase

- U jeziku C++ klasa se deklariše naredbom koja ima opšti oblik

```
class naziv {Lista_članova} Lista_objekata;
```

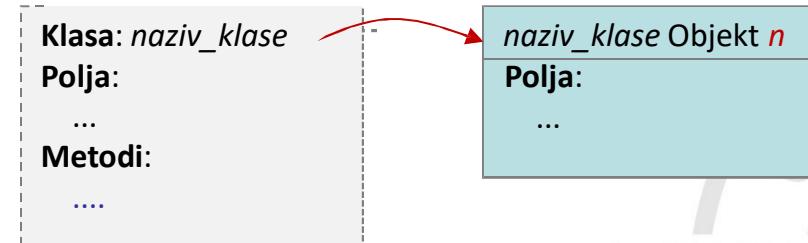
 - naziv klase je novi korisnički definisani tip podataka
 - lista članova je spisak članova klase - podataka (*member data*) i funkcija (*member functions*).
Članovi klase mogu biti *privatni* i *javni*.
Privatnim članovima pristupaju samo drugi članovi klase, a javnim i drugi delovi programa. Podrazumevajući je privatni pristup; za javni pristup članovima se navodi ključna reč public
 - lista objekata (nije obavezna) je spisak naziva *objekata* novog tipa
- Načelno, klasa treba da predstavlja logičku celinu i grupiše samo logički povezane informacije

Deklarisanje promenljive tipa klase

- Klasa predstavlja novi **tip** podataka koji se može koristiti za kreiranje objekata - instanci klase
- Deklaracija **promenljive** tipa klase

naziv_klase naziv_promenljive;

prouzrokuje *kreiranje objekta* za koji se rezerviše memorijski prostor



- Članovima klase pristupa se pomoću naziva objekta i operatora tačka

objekt.član;

2.2 Enkapsuliranje klase

- Apstrakcija klase je izdvajanje i predstavljanje samo bitnih svojstava objekata
- Enkapsuliranje je *skrivanje* od korisnika detalja načina objedinjavanja podataka i metoda u jednu celinu, odnosno implementacije klase
 - razdvajanje implementacije klase od njene upotrebe
 - članovi klase navedeni kao **private** zaštićeni su od pristupa izvan same klase (skriveni ili enkapsulirani)
 - u stvarnosti su detalji implementacije različitih pojava/sistema skriveni od korisnika; npr. *tehnički sistemi*, kao što su računari i uređaji u domaćinstvu, za koje korisnici znaju samo funkcije i način njihove upotrebe ili *bankarski krediti*, za koje korisnici znaju uslove, rokove i efekte, dok ih detalji procesa odobravanja i obrade kredita ne zanimaju

Primer: Upotreba klasa u programu (1/2)

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

// Deklarisanje klase bez funkcija članova
class Vozilo {
public:
    int brojMesta;
    int kapacitetRezervoara;
    int potrosnja;
};

int main() {
    // Deklarisanje promenljivih
    Vozilo kombi;
    Vozilo automobil;
    double autonomijaKombi, autonomijaAuto;
    // Dodela vrednosti podacima članovima objekta kombi
    kombi.brojMesta = 7;
    kombi.kapacitetRezervoara = 60;
    kombi.potrosnja = 7.;
```

obavezno na kraju naredbe deklarisanja klase

Primer: Upotreba klasa u programu (2/2)

```
// Dodela vrednosti podacima članovima objekta automobil
automobil.brojMesta = 5;
automobil.kapacitetRezervoara = 50;
automobil.potrosnja = 5.;

autonomijaKombi = kombi.kapacitetRezervoara / kombi.potrosnja*100;
autonomijaAuto = automobil.kapacitetRezervoara / automobil.potrosnja*100;

cout << "U kombi staje " << kombi.brojMesta
    << " putnika. S punim rezervoarom prelazi "
    << autonomijaKombi << " kilometara." << endl;

cout << "U automobil staje " << automobil.brojMesta
    << " putnika. S punim rezervoarom prelazi "
    << autonomijaAuto << " kilometara." << endl;
return 0
}
```

Izvršavanje programa:

```
U kombi staje 7 putnika. S punim rezervoarom prelazi 800 kilometara.
U automobil staje 5 putnika. S punim rezervoarom prelazi 1000 kilometara.
```

2.3 Definisanje funkcija članova klase

- Funkcije članovi klase zadaju se pomoću *prototipa*, npr.

```
class Vozilo {  
public:  
    int brojMesta;  
    int kapacitetRezervoara;  
    int potrosnja;  
    int autonomija(); // funkcija član klase  
}
```

- Implementacija funkcije člana navodi se zasebno, uz upotrebu operatora razrešenja dosega ::

```
// Implementacija funkcije člana: autonomija  
int Vozilo::autonomija() {  
    return kapacitetRezervoara / potrosnja*100; //direktni pristup čl.  
}
```

2.4 Pristup članovima klase

- Članovi klase su **podaci i metodi** (funkcije), kojima se pristupa pomoću naziva objekta i operatora tačka , npr.
objekt.polje;
objekt.funkcija(...);
- Pristup članovima klase definiše se pomoću specifikacija pristupa **public:** i **private:**
- Podrazumeva se *privatni* pristup, samo za funkcije članove klase, koji je direktni, bez operatora tačka
- Ipak, dobra praksa programiranja je da se specifikacije pristupa navode *eksplicitno* i to prvo privatni, a zatim javni članovi klase

Primer: Upotreba javnih polja klase

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

// Deklarisanje klase
class MojaKlasa {
    public: int i, j, k;
};

int main() {
    // Deklarisanje promenljivih
    MojaKlasa a, b;
    // Polja i, j, k su vidljiva
    a.i = 100;
    a.j = 4;
    a.k = a.i * a.j;
    b.k = 12;
    cout << a.k << " " << b.k; // ispis: 400 12
}
```



2.5 Ugrađene (inline) funkcije

- Moguće je definisati kompletnu funkciju unutar deklaracije klase

```
class Brojac {  
    private:  
        int i;  
    public:  
        int broj { return ++i; } // inline funkcija član  
}
```

- Takva funkcija se naziva **ugrađena (inline) funkcija**. Prevodilac *ugrađuje kod* takve funkcije na mestima gde se ona poziva
- Ugrađena funkcija se može definisati i izvan deklaracije klase:

```
inline int f () { ... }
```

3. Konstruktori i destruktur

1. Pojam konstruktora i destruktora
2. Upotreba konstruktora i destruktora
3. Parametri konstruktora
4. Preklapanje konstruktora
5. Podrazumevani konstruktori
6. Konstruktor kopije



3.1 Pojam konstruktora i destruktora

- Konstruktor je funkcija član klase koja se automatski poziva prilikom kreiranja objekta, obično radi dodele početnih vrednosti članovima klase
- U jeziku C++ konstruktor je funkcija istog naziva kao i klasa, koja ne vraća rezultat i za koju tip rezultata nije definisan
- Sintaksa definicije konstruktora je:

```
naziv_klase () {  
    // telo konstruktora  
}
```



Pojam konstruktora i destruktora

- Destruktor je funkcija član klase koja se automatski poziva radi izvođenja različitih operacija prilikom uništavanja objekta
- Destruktor je funkcija naziva kao i klasa s dodatkom ~ ispred, koja ne vraća rezultat i za koju tip rezultata nije definisan
- Sintaksa definicije destruktora je:

```
~naziv_klase () {  
    // telo destruktora  
}
```

- Klasa ima samo jedan destruktur; ako se posebno ne navede, prevodilac će ga kreirati automatski

Primer: Definicija konstruktora i destruktora klase

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

// Deklaracija klase
class Primer {
public:
    Primer();      // prototip konstruktora
    ~Primer();    // prototip destruktora
};

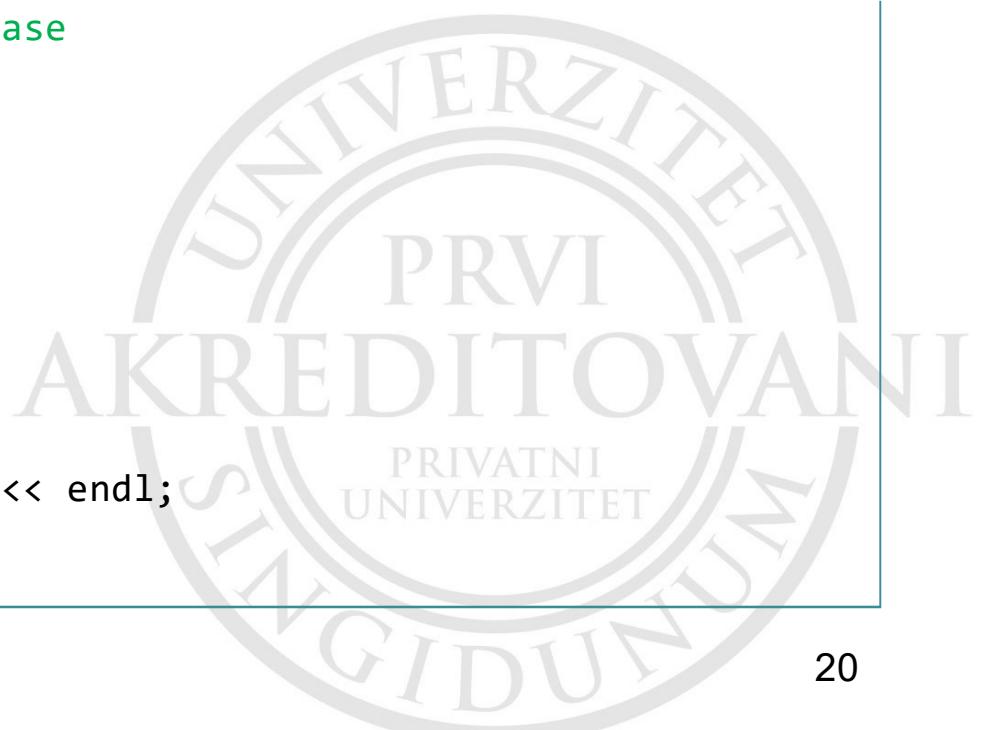
// Definicija konstruktora i destruktora klase
Primer::Primer(){
    cout << "u konstruktoru" << endl;
}

Primer::~Primer(){
    cout << "u destruktoru" << endl;
}

int main() {
    Primer obj;    // pokreće konstruktor
    cout << "Program koji demonstrira objekt" << endl;
    return 0;      // pokreće destruktora
}
```

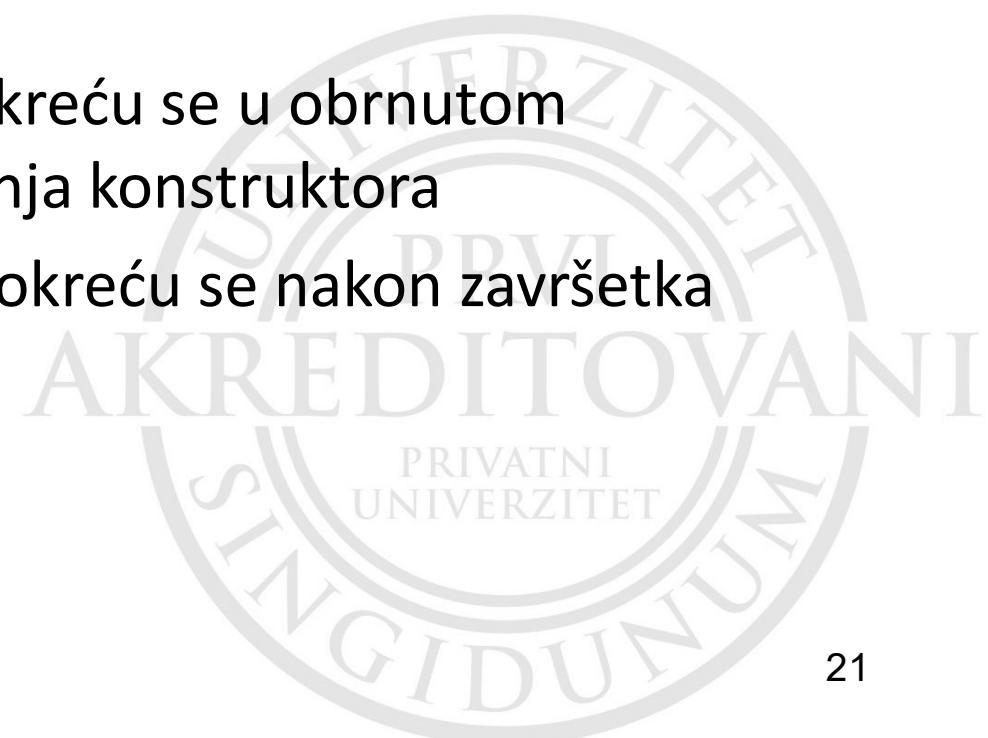
Izvršavanje programa:

u konstruktoru
Program koji demonstrira objekt
u destruktoru



3.2 Upotreba konstruktora i destruktora

- U jeziku C++ konstruktori globalnih objekata pokreću se na početku programa, pre početka izvršavanja funkcije **main()**, po redosledu deklarisanja
- Deklaracija svake lokalne promenljive pokreće konstruktor lokalnog objekta
- Destruktori lokalnih objekata pokreću se u obrnutom redosledu od redosleda pokretanja konstruktora
- Destruktori globalnih objekata pokreću se nakon završetka funkcije **main()**



Primer: Redosled izvršavanja konstruktora i destruktora

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

class MojaKlasa {
public:
    int koji;
    MojaKlasa(int id);
    ~MojaKlasa();
} glob_obj1(1), glob_obj2(2); // kreirana dva objekta

MojaKlasa::MojaKlasa(int id) {
    cout << "Inicijalizuje se " << id << endl;
    koji = id;
}

MojaKlasa::~MojaKlasa() {
    cout << "Unistava se " << koji << endl;
}

int main() {
    MojaKlasa lokalni_obj1(3); // kreirana objekt 3
    cout << "Ovo nece biti prvi prikazani red." << endl;
    MojaKlasa lokalni_obj2(4); // kreirana objekt 4
    return 0;
}
```

Izvršavanje programa:

```
Inicijalizuje se 1
Inicijalizuje se 2
Inicijalizuje se 3
Ovo nece biti prvi prikazani red.
Inicijalizuje se 4
Unistava se 4
Unistava se 3
Unistava se 2
Unistava se 1
```

3.3 Parametri konstruktora

- Konstruktori su funkcije koje mogu imati parametre, koji prenose vrednosti za inicijalizaciju novog objekta, npr.

`MojaKlasa obj(3, 5);`

odgovara naredbi

`MojaKlasa obj = mojaKlasa(3, 5);`

- Parametri konstruktora mogu imati podrazumevane vrednosti
- Ako konstruktor ima samo jedan parametar, za dodelu vrednosti se može koristiti *inline* funkcija konstruktor

`MojaKlasa(int param) { promenljiva = param; } // inline k.`

`int get_promenljiva() { return promenljiva; } // pristup`

a vrednost se novom objektu dodeljuje kao

`MojaKlasa obj = vrednost;`

Primer: Parametri konstruktora

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

class MojaKlasa {
    int a, b;
public:
    // Inline konstruktor s parametrima
    MojaKlasa(int i, int j) {
        a = i;
        b = j;
    }
    void prikazi(){
        cout << a << " " << b << endl;
    }
};

int main() {
    MojaKlasa obj(3, 5);
    obj.prikazi();
    return 0;
}
```

Izvršavanje programa:

3 5



Primer: Podrazumevane vrednosti konstruktora

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

class Osoba {
    char *imePrezime;
    int godine;
public:
    Osoba(char *ime="Petar Petrovic", int godine=21); // podrazumevane vrednosti konstruktora
    void koJeOsoba();
};

// Konstruktor
Osoba::Osoba(char *i, int g) {
    imePrezime = i; godine = g;
}

void Osoba::koJeOsoba() {
    cout << imePrezime << endl;
}

int main() {
    Osoba o;          // Konstruktor s podrazumevanim vrednostima
    o.koJeOsoba();   // Ispisuje podrazumevanu vrednost Petar Petrovic
    return 0
}
```

Izvršavanje programa:

Petar Petrovic



3.4 Preklapanje konstruktora

- Prilagodljivost klasa različitim potrebama i tipovima objekata obezbeđuje se preklapanjem konstruktora
- Za svaki od mogućih načina kreiranja objekta obezbedi se poseban konstruktor
- Primer je unos datuma, koji se može zadati na dva načina:
 - u obliku tri cela broja (*dan, mesec, godina*) ili
 - kao datumski string *dd.mm.gggg*
- Klasa **datum** može da prihvati oba načina inicijalizacije ako se predvide dva (preklopljena) konstruktora

Primer: Preklapanje konstruktora

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

class Datum {
    int dan, mesec, godina;
public:
    Datum(char *d);
    Datum(int d, int m, int g);
    void prikazi_datum();
};

// Inicijalizacija za vrednosti tipa string
Datum::Datum(char *d) {
    sscanf(d, "%d.%d.%d", &dan, &mesec, &godina);
}

// Inicijalizacija za celobrojne vrednosti
Datum::Datum(int d, int m, int g) {
    dan = d; mesec = m; godina = g;
}
```

```
void Datum::prikazi_datum() {
    cout << dan << "." << mesec << "." <<
        godina << endl;
}

int main() {
    // Inicijalizacija datuma na dva načina
    Datum ob1(24, 3, 2020),
        ob2("24.3.2020");
    ob1.prikazi_datum();
    ob2.prikazi_datum();
    return 0;
}
```

Izvršavanje programa:

```
24.3.2020
24.3.2020
```

sscanf: čitanje podataka iz stringa s
 Specifikacija formata (d - cifra): %[*][width][length]specifier

3.5 Podrazumevani konstruktori

- Ako se klasa deklariše bez eksplisitno opisanog konstruktora, prevodilac će generisati **podrazumevani (default) konstruktor**, koji samo kreira objekt klase, *bez inicijalizacije* podataka članova klase
- Primer
 - deklaracija objekta klase bez argumenata iz prethodnog primera
`Datum d;`
izazvala bi grešku. Za ovu vrstu deklaracije objekta, u primeru bi klasi `Datum` trebalo dodati još jedan konstruktor, koji *nema argumente*
...
`Datum(); // default konstruktor klase datum`
...

3.6 Konstruktor kopije

- Konstruktor kopije (*copy constructor*) je posebna vrsta konstruktora za *inicijalizaciju* objekta drugim objektom

- Problem je što standardna dodela vrednosti

MojaKlase B = A; // objekt B se kreira kao identična kopija A

kreira vernu kopiju objekta A, tako da objekt B koristi istu memoriju kao i objekt A i *iste pokazivače* na promenljive u memoriji koju će destruktor *jednog* od objekata osloboditi

- Da se to izbegne kreira se poseban konstruktor za **kopije objekta**, koji definiše nepromenljivu *referencu* na objekt

```
naziv_klase (const naziv_klase &objekt ...) {  
    // telo konstruktora  
}
```

Dodela vrednosti i inicijalizacija

- Dodela vrednosti objekta drugom objektu različito se tretira kad je u pitanju standardna dodata vrednosti i inicijalizacija
- Inicijalizacija može biti
 - inicijalizacija jednog objekta drugim
 - kreiranje privremenog objekta
 - kopija objekta kao argumenta funkcije
- Konstruktor kopije se koristi *samo za inicijalizaciju*, dok se u dodeli vrednosti ne koristi, npr.

```
MojaKlasa x=y; // inicijalizacija koristi konstruktor kopije
f(y);           // argument koristi konstruktor kopije
MojaKlasa a, b;
a = b;          // dodela ne koristi konstruktor kopije!
```

Ilustracija: Upotreba konstruktora kopije

```
class Osoba {  
    char *ime;  
    int godine;  
  
public:  
    Osoba(char *i, int g) {          // konstruktor  
        ime = new char[strlen(i)+1];  
        strcpy(ime, i);  
        godine = g;  
    }  
    Osoba(const Osoba &osoba) {      // konstruktor kopije  
        ime = new char[strlen(osoba.ime)+1];  
        strcpy(ime, osoba.ime);  
        godine = osoba.godine;  
    }  
    ~Osoba() { delete[] ime; }       // destruktur  
};
```



4. Upotreba objekata

1. Zajednički članovi klase
2. Objekti kao parametri funkcije
3. Dodela vrednosti objekata
4. Pokazivači na objekte
5. Pokazivač this



4.1 Zajednički članovi klase

- Deklaracija **static** ispred podatka člana klase znači da postoji samo jedna kopija tog podatka za sve kreirane objekte klase
- Takav podatak je *zajednički podatak član*, koji dele svi objekti klase
- Sve statičke promenljive klasa, odnosno zajednički podaci članovi, deklarišu se izvan klase i inicijalizuje na **0** pre kreiranja prvog objekta
- Statičke promenljive se koriste za pristup nekom deljenom resursu, npr. istom fajlu ili za praćenje ukupnog broja objekata neke klase

Primer: Upotreba statičkog podatka člana

```
# include <iostream>
using namespace std;
class Deljena {
    static int a;
    int b;
public:
    void set(int i, int j) {a=i; b=j;}
    void prikazi();
};

int Deljena::a; // pristup statičkom
                // članu i bez objekta
void Deljena::prikazi() {
    cout << "Staticko a: " << a << endl;
    cout << "Nestaticko b: " << b <<
        endl;
}
```

```
int main() {
    Deljena x, y;
    x.set(1, 1); // a poprima vrednost 1
    x.prikazi();
    y.set(2, 2); // a poprima vrednost 2
    y.prikazi()
    // (statički) a je promenjen i za x
    x.prikazi()
    return 0;
}
```

Izvršavanje program a:

```
Staticko a: 1
Nestaticko b: 1
Staticko a: 2
Nestaticko b: 2
Staticko a: 2
Nestaticko b: 1
```

Primer: Upotreba statičkog podatka člana za praćenje ukupnog broja objekata

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

class Brojac {
public:
    static int broj;
    Brojac() { broj++; } // konstruktor
    ~Brojac() { broj--; } // destruktor
};

int Brojac::broj; // pristup statičkom
                  // članu bez objekta

void f() {
    Brojac temp; // konstruktor (++)
    cout << "Ukupno objekata: ";
    cout << Brojac::broj << endl;
    // temp se unistava nakon zavrsetka f
    // destruktor (--)
}
```

```
int main() {
    Brojac o1;      // konstruktor (++)
    cout << "Ukupno objekata: ";
    cout << Brojac::broj << endl;
    Brojac o2;      // konstruktor (++)
    f();
    cout << "Ukupno objekata: ";
    cout << Brojac::broj << endl;
    return 0;
}
```

Izvršavanje programa:

```
Ukupno objekata: 1
Ukupno objekata: 2
Ukupno objekata: 3
Ukupno objekata: 2
```

Funkcija kao zajednički član klase

- Funkcije se mogu deklarisati kao *statičke*
 - pristupaju samo statičkim poljima i mogu se koristiti za inicijalizaciju *privatnih* statičkih podataka članova klase
 - klasa može da ima dve verzije funkcije istog imena, *statičke* i *nestatičke*

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

class Staticka {
    static int i;
public:
    static int broj;
    static void init(int x) {i = x;}
    void prikazi() {cout << i << endl;}
};

int Staticka::i; // definiše se i

int main() {
    // Inicijalizacija statičkih podataka
    // pre kreiranja objekata
    Staticka::init(100);
    Staticka x; // kreiranje objekta x
    x.prikazi(); // prikazuje 100
    return 0;
}
```

4.2 Objekti kao parametri funkcije

- Objekti se mogu prosleđivati funkcijama *po vrednosti*, kada se kao argument prenosi *kopija* objekta (novi objekt)
- Prilikom kreiranja novog objekta ne pokreće se konstruktor (kreira se verna kopija objekta, bez inicijalizacije), ali se za uništavanja objekta pokreće destruktur da oslobodi memoriju
 - problem: npr. kada objekt koji se prenosi kao argument alocira i dealocira memoriju
Memoriju može da oslobodi i destruktur kopije objekta i time originalni objekt učini neupotrebljivim
Zbog toga se u takvim slučajevim se definiše poseban *konstruktor kopije*

Primer: Konstruktor i destruktur objekta argumenta funkcije

```
# include <iostream>
using namespace std;
class MojaKlasa {
    int i;
public:
    MojaKlasa(int n);
    ~MojaKlasa();
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};
MojaKlasa::mojaKlasa(int n) {
    i = n;
    cout << "Kreira se " << i << endl;
}
MojaKlasa::~mojaKlasa() {
    cout << "Unistava se " << i << endl;
}
void f(mojaKlasa obj);
```

```
int main() {
    MojaKlasa o(1);
    f(o);
    cout << "i u main(): ";
    cout << o.get_i() << endl;
    return 0;
}

void f(mojaKlasa obj) {
    obj.set_i(2);
    cout << "i u funkciji f(): "
        << obj.get_i() << endl;
}
```

Izvršavanje programa:

```
Kreira se 1
i u funkciji f(): 2
Unistava se 2
i u main(): 1
Unistava se 1
```

4.3 Dodela vrednosti objekata

- Objekti istog tipa mogu se dodeljivati jedan drugome naredbom dodele vrednosti
- Podaci objekta s desne strane kopiraju se u podatke objekta s leve strane
- Koji je rezultat izvršavanja programa na slici?

Izvršavanje programa:

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

class MojaKlasa {
    int i;
public:
    void set_i(int n) { i = n; }
    int get_i() { return i; }
};

int main() {
    MojaKlasa obj1, obj2;
    obj1.set_i(99);
    obj2 = obj1; // dodela objekta
    cout << "obj2.i: "
        << obj2.get_i() << endl;
    return 0;
}
```

4.4 Pokazivači na objekte

- Pristup objektima može se ostvariti preko *pokazivača*, kada se pristup članovima objekta ostvaruje pomoću *operatora ->*
- Npr. pristup funkciji članu se označava kao *p->funkcija()*, što je ekvivalentno notaciji *(*p).funkcija()*
 - zagrada je potrebna zbog višeg prioriteta operatora *.* od operatora ***
- Operator adresiranja **&** koristi se i za objekte, kao i reference

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

class MojaKlasa {
    int i;
public:
    MojaKlasa(int j) { i = j; }
    int get_i() { return i; }
};

int main() {
    MojaKlasa obj(88), *p;
    p = &obj; // p pokazuje na obj
    cout << p->get_i() << endl; // 88
    return 0;
}
```

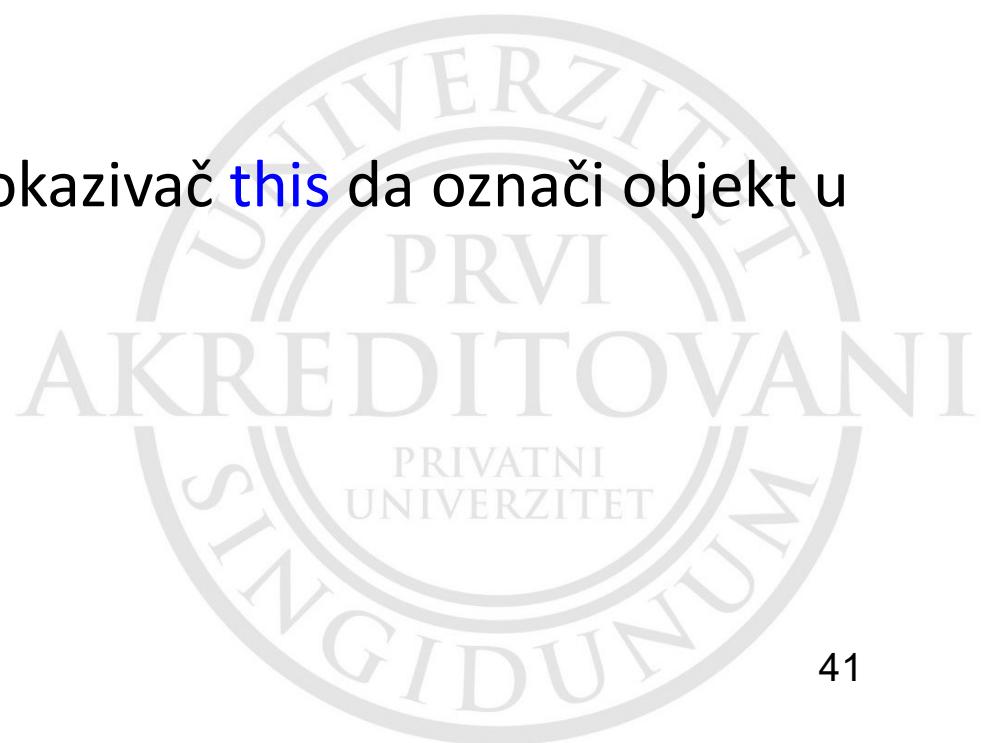
4.5 Pokazivač *this*

- Funkcije članovi klase pozivaju se za sve objekte neke klase
- Prilikom poziva, funkciji se implicitno šalje pokazivač na objekt za koji je pokrenuta, čiji je naziv **this**
- Uvek kad funkcija član klase koristi naziv nekog podatka člana, npr.

b = vrednost;

prevodilac automatski koristi pokazivač **this** da označi objekt u kome se nalazi promenljiva:

this->b = vrednost;



Primer: Implicitni pokazivač *this*

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

class Pwr {
    double b;
    int e;
    double val;
public:
    Pwr(double base, int exp);
    double get_pwr() { return val; }
};

Pwr::Pwr(double base, int exp) {
    b = base; // this->b = base;
    e = exp; // this->e = exp;
    val = 1; // this->val = 1;
    if (exp==0)
        return;
    for(; exp>0; exp--)
        val = val * b;
    // this->val = this->val * this->b;
}
```

```
int main() {
    Pwr x(4.0, 2), y(2.5, 1), z(5.7, 0);
    cout << x.get_pwr() << " ";
    cout << y.get_pwr() << " ";
    cout << z.get_pwr() << endl;
    return 0;
}
```

Izvršavanje programa:

16 2.5 1



5. Klasa string

- Upotreba klase string
- Inicijalizacija objekata klase string
- Funkcije klase string



Upotreba klase string

- Klasa **string** je ugrađena u jezik C++ preko programske biblioteke koja se aktivira pomoću datoteke zaglavlja **<string>**

- objekti klase string koriste se kao objekti osnovnih klasa, npr.

```
string ime;                                // deklarisanje
cout << "Unesite ime: ";
cin << ime;                                // ucitavanje
cout << "Dobar dan, " << ime << endl; // ispis
```

- učitavanje celog reda teksta klase string (s belinama) vrši se funkcijom **getline(tok_objekt, string_objekt)**

- poređenje stringova ove klase vrši se operatorima **<**, **<=**, **>**, **>=** i **==**. Konkatenacija stringova vrši se operatorima **+** i **+=**, npr.

```
string s1 = "ABC";
s2 = "DEF";
s3 = s1 + s2;
```

Inicijalizacija objekata klase string

string adresa;	Deklariše prazan string adresa
string ime("Nikola Vukovic");	Deklariše string objekat ime i inicijalizuje ga sa "Nikola Vukovic"
string osoba2(osoba1)	Deklariše string objekat osoba1, koji je kopija string objekta osoba2, pri čemu osoba2 može da bude drugi string ili niz znakova
string set1(set2, 5);	Deklariše string objekat set1 koji se inicijalizuje sa prvih pet znakova niza znakova set2
string punRed ('z', 10);	Deklariše string objekat punRed koji se inicijalizuje sa 10 znakova 'z'
string ime(punoIme, 0, 7);	Deklariše string objekat ime koji se inicijalizuje podstringom stringa punoIme. Podstring ima 7 znakova i počinje od pozicije 0.

Funkcije klase string (1/2)

- Deo funkcija članica klase **string** naveden je u tabeli

<code>str1.append(str2)</code>	Dodaje str2 na str1. str2 može biti string objekat ili niz znakova.
<code>str1.append(str2, x, n)</code>	n znakova objekta str2 počev od pozicije x dodaju se objektu str1. Ako str1 nema dovoljnu dužinu, kopiraće se onoliko znakova koliko može da stane.
<code>str1.append(str2, n);</code>	Prvih n znakovâ niza str2 dodeljuju se str1
<code>str.append(n, 'z')</code>	n kopija znaka 'z' dodeljuje se objektu str
<code>str1.assign(str2)</code>	Dodeljuje str2 objektu str1. str2 može da bude string objekat ili niz znakova.
<code>str1.assign(str2, x, n)</code>	n znakova objekta str2 počev od pozicije x dodeljuje se str1. Ako str1 nema dovoljnu dužinu, kopiraće se onoliko znakova koliko može da stane.
<code>str1.assign(str2, n)</code>	Prvih n znakova str2 dodeljuje se objektu str1.
<code>str.assign(n, 'z')</code>	Dodeljuje n kopija znaka 'z' objektu str.
<code>str.at(x)</code>	Vraća znak na poziciji x u objektu str.

Funkcije klase string (2/2)

<code>str.capacity()</code>	Vraća veličinu memorije koja je alocirana za string.
<code>str.clear()</code>	Briše sve znakove u stringu str.
<code>str1.compare(str2)</code>	Poredi stringove kao funkcija <code>strcmp</code> za C stringove, sa istim povratnim vrednostima. str2 može da bude niz znakova ili string objekat.
<code>str1.compare(x, n, str2)</code>	Poredi stringove str1 i str2 počev od pozicije x narednih n znakova. Povratna vrednost je ista kao u funkciji <code>strcmp</code> . str2 može da bude string objekat ili niz znakova.
<code>str1.copy(str2, x, n)</code>	Kopira n znakova niza znakova str2 u str1 počev od pozicije x. Ako str2 nije dovoljno dugačak, funkcija kopira onoliko znakova koliko može da stane.
<code>str.data()</code>	Vraća niz znakova sa nulom na kraju, kao u str
<code>str.empty()</code>	Vraća true ako je str prazan.
<code>str.erase(x, n)</code>	Briše n znakova iz objekta str počev od pozicije x.

<code>str1.find(str2, x)</code>	Vraća prvu poziciju iza pozicije x gde se string str2 nalazi u str1. str2 može da bude string objekat ili niz znakova.
<code>str.find('z', x)</code>	Vraća prvu poziciju iza pozicije x na kojoj se znak 'z' nalazi u str1
<code>str1.insert(x, str2)</code>	Umeće kopiju str2 u str1 počev pod pozicije x. str2 može da bude string objekat ili niz znakova.
<code>str.insert(x, n, 'z')</code>	Umeće 'z' n puta u str počev od pozicije x
<code>str.length()</code>	Vraća dužinu stringa str
<code>str1.replace(x, n, str2)</code>	Zamenjuje n znakova u str1 počev od pozicije x znakovima iz string objekta str2
<code>str.size()</code>	Vraća dužinu stringa str
<code>str.substr(x, n)</code>	Vraća kopiju podstringa dugačkog n znakova koji počinje na poziciji x objekta str
<code>str1.swap(str2)</code>	Zamenjuje sadržaj str1 sa str2



6. Nasleđivanje i izvedene klase

1. Nasleđivanje u klasama
2. Konstruktori u nasleđenoj klasi
3. Destruktori i nasleđivanje
4. Duplikati naziva članova klase
5. Višestruko nasleđivanje
6. Konverzija tipova povezanih klasa

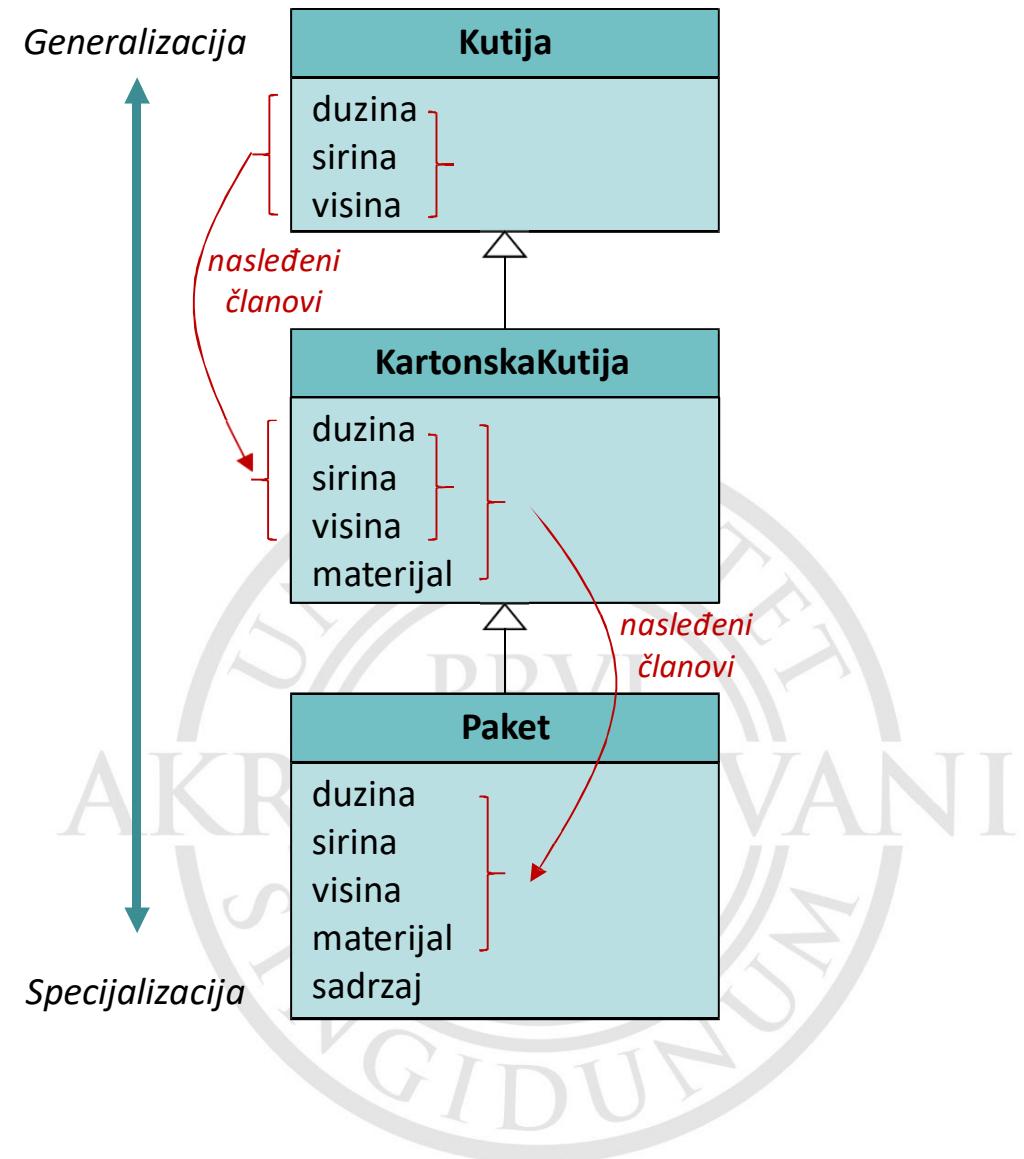


6.1 Nasleđivanje u klasama

- Nasleđivanje je način da se nove klase kreiraju ponovnom upotrebom i proširenjem definicija postojećih klasa
- Klase modeliraju skupove entiteta koji imaju neke *zajedničke* osobine i ponašanje i mogu biti međusobno povezane na različite načine, kao i stvarni entiteti
- Objekti nekih klasa mogu biti sastavni elementi objekata drugih klasa ili mogu biti specijalni slučaj objekata drugih klasa, npr.
 - klase **Motor** i **Automobil** (motor je *sastavni deo* automobila)
 - klase **Pas** i **Sisar** (pas je *specijalni slučaj* sisara)

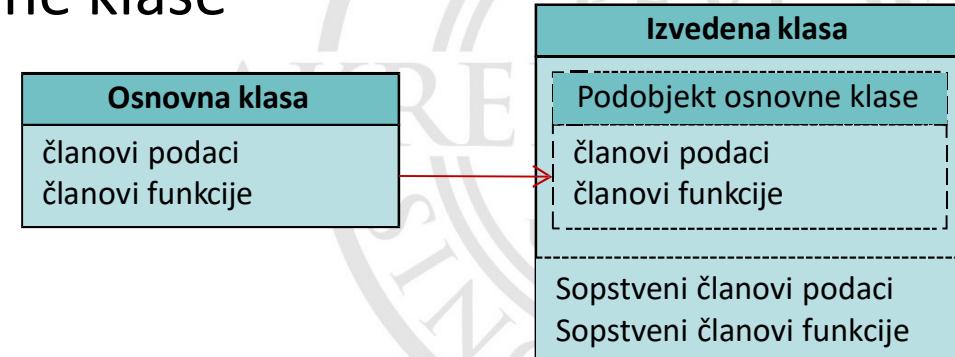
Hijerarhije klasa

- Između klasa može postojati višestruka veza generalizacije/specijalizacije (hijerarhija)
- Npr. klasa **KartonskaKutija** je *izvedena* iz osnovne klase **Kutija**, a klasa **Paket** iz osnovne klase **KartonskaKutija**
- U primeru su *izvedene* klase nasledile sve osobine *osnovnih* klasa



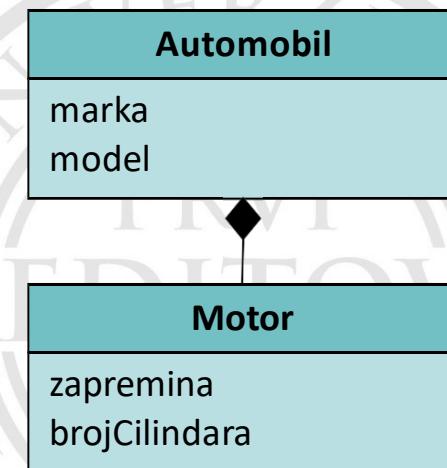
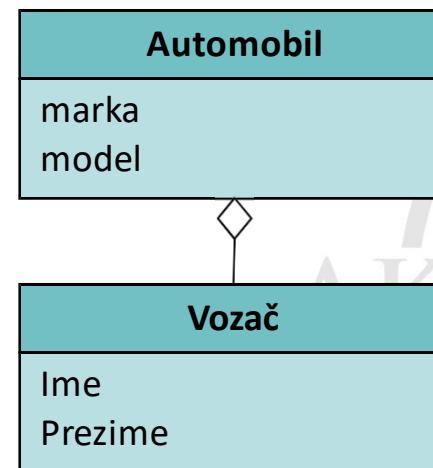
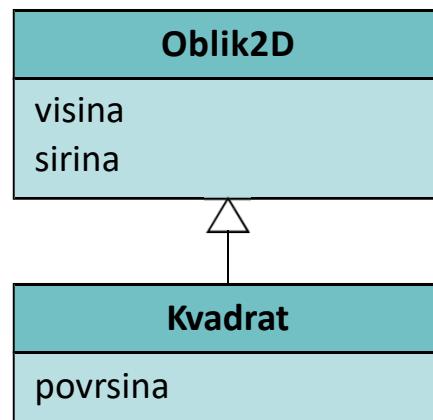
Izvedene klase

- Nasleđivanje omogućava novoj klasi da nasledi osobine neke druge postojeće klase
- Izvedena klasa je podklasa (*subclass*), a osnovna klasa je nadklasa (*superclass*)
- Nasleđivanje može biti *direktno* ili *indirektno*, preko druge izvedene klase
- Izvedena klasa sadrži sve članove podatke i (uz ograničenja) članove funkcije iz osnovne klase
- Izvedena klasa sadrži i sopstvene članove, podatke i funkcije



Nasleđivanje i agregacija

- Izvedena klasa može predstavljati npr. specijalni slučaj osnovne klase (*is-a*) ili može biti njen sastavni deo (*has-a*)
- Odgovarajuće veze između klasa nazivaju se *generalizacija* (Δ) i *agregacija* (\lozenge) ili *kompozicija* (\blacklozenge)
 - zavisno od toga da li se sastavni deo može izostaviti ili je obavezan



Izvođenje klase

- Sintaksa izvođenja klase je

```
class Izvedena_klase : specifikacija Osnovna_klase {  
    // Telo izvedene klase  
}
```

- *Specifikacija* izvođenja nije obavezna i može biti **public**, **private** ili **protected**
- Ako se izostavi, podrazumeva se specifikacija izvođenja **public**, koja označava da će svi javni članovi osnovne klase takođe biti javni u izvedenim klasama
- Izvedene klase omogućavaju lokalizaciju specifičnih svojstava nove klase u telu klase, čime se pojednostavljuje održavanje programa

Primer: Izvođenje klase (specijalizacija)

```
class Oblik2D {  
public:  
    double visina;  
    double sirina;  
    void prikaziDimenzije() {  
        cout << "sirina= " << sirina << " visina= " << visina << endl;  
    }  
};  
  
class Trougao: public Oblik2D {  
public:  
    string vrsta;  
    double povrsina() {  
        return sirina * visina / 2;  
    }  
};
```

nasleđeni članovi:
visina, sirina, prikaziDimenzije()



Zaštićeni članovi klase

- Podrazumevajuća specifikacija izvođenja **public** dozvoljava izvedenoj klasi pristup svim javnim članovima osnovne klase
- Pristup izvedene klase članovima osnovne klase može se *zabraniti* pomoću specifikacije izvođenja **private**
- Pristup izvedene klase članovima osnovne klase, uz istovremenu zaštitu od pristupa drugih klasa, može se omogućiti korišćenjem specifikacije izvođenja **protected**, npr.

```
class Kutija {  
    protected:  
        double duzina; double sirina; double visina;  
    public:  
        ...  
}
```

Primer: Pristup zaštićenim članovima klase

```
class Osnovna {  
    private:  
        int privatni;  
    protected:  
        int zasticen; → samo članovi osnovne i izvedene  
    public:  
        int javni;  
};  
  
class Izvedena: public Osnovna {  
public:  
    void prikazi() {  
        cout << zasticen << endl;  
        cout << javni << endl;  
        cout << privatni << endl; // greška!  
    }  
};
```

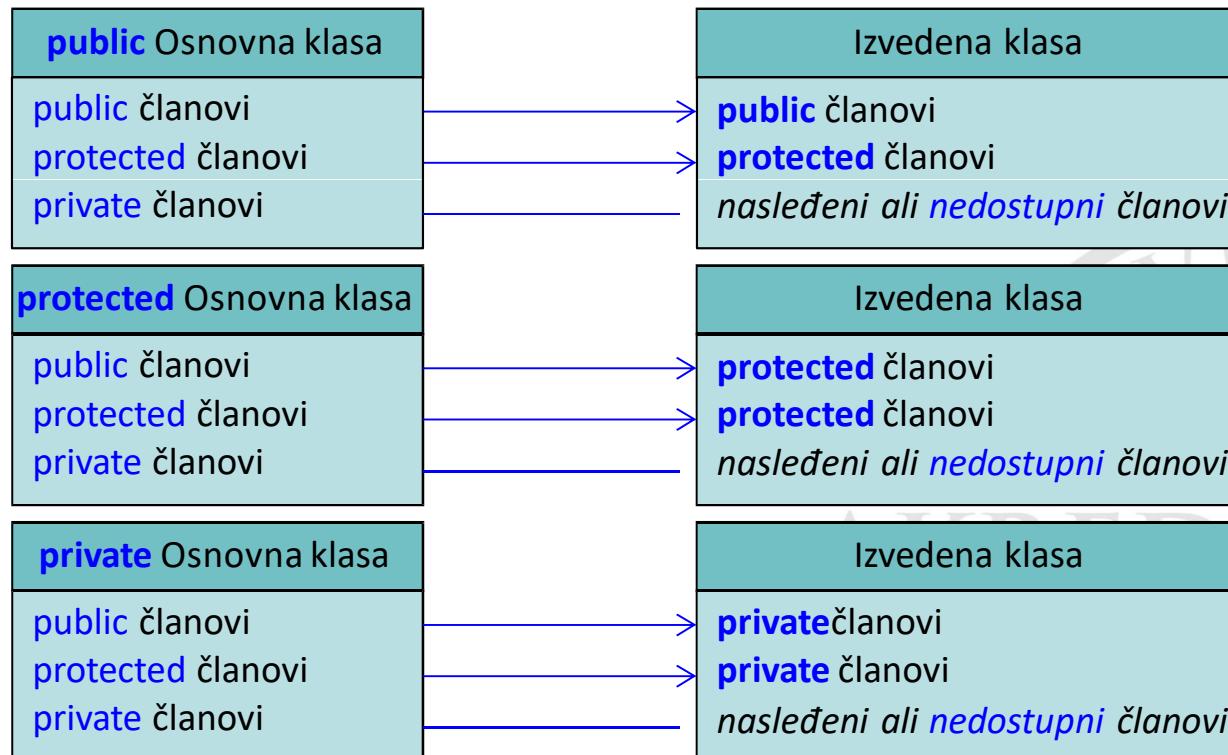


Specifikacija pristupa

- Izvođenjem klase dostupnost podataka u izvedenoj klasi može biti samo na istom ili nižem nivou
 - prilikom izvođenja klase iz osnovne specifikatorom pristupa **public** zaštićeni članovi ostaju zaštićeni i u izvedenoj klasi
 - kada se klasa izvodi specifikatorom **private**, zaštićeni članovi osnovne klase postaju **private** u izvedenoj klasi
- **Javno** izvođenje omogućava pristup javnim članovima osnovne klase i u izvedenoj klasi, tako da je izvedena klasa vrsta (*a kind of*) osnovne klase
- **Privatno** izvođenje skriva članove osnovne klase u izvedenoj klasi, tako da je izvedena klasa deo (*a part of*) osnovne klase, odnosno opisuje članstvo objekta ili *kompoziciju*

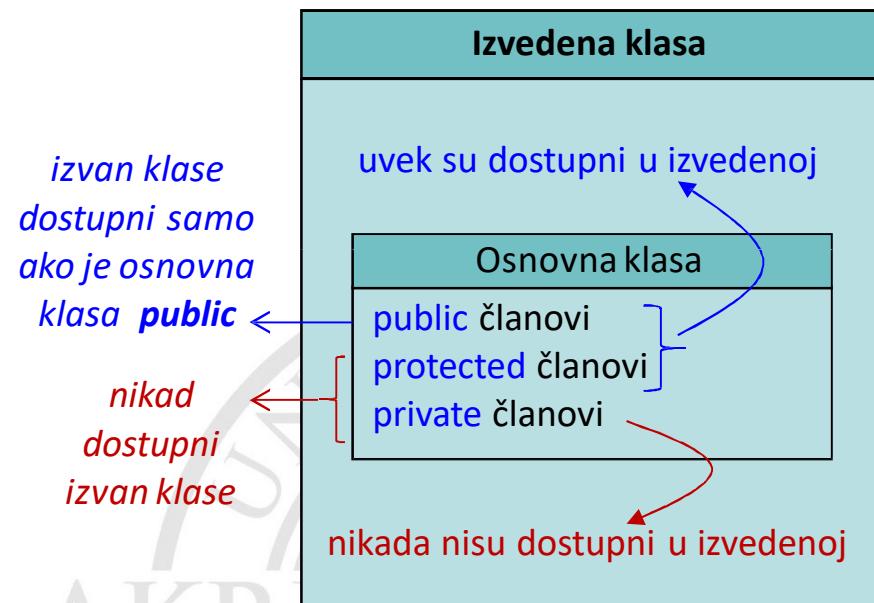
Pregled specifikacija pristupa

- Pristup članovima klase *istovremeno* zavisi od specifikacije pristupa u *osnovnoj* i *izvedenoj* klasi:



Izbor specifikacije pristupa u hijerarhijama klasa

- Javni (**public**) i zaštićeni (**protected**) članovi osnovne klase uz *specifikaciju* ...
 - ... **public** ostaju *javni* i *zaštićeni* u izvedenoj klasi
 - ... **protected** postaju *zaštićeni* članovi izvedene klase
 - ... **private** postaju *privatni* članovi izvedene klase
- Privatni (**private**) članovi osnovne klase se nasleđuju, ali su tada *uvek nedostupni*



6.2 Konstruktori u nasleđenoj klasi

- Konstruktor izvedene klase
- Prenos argumenata konstruktoru osnovne klase
- Nasleđeni konstruktori



Konstruktor izvedene klase

- Konstruktor osnovne klase kreira deo objekta koji pripada osnovnoj klasi, a deo koji pripada izvedenoj klasi kreira konstruktor izvedene klase
- Kada su *u obe klase* definisani konstruktori, izvedena klasa mora eksplicitno da pokrene konstruktor osnovne klase, npr.

```
Naziv_izvedene_klase (Lista_argumenata) :
```

```
    Konstruktor_osnovne_klase(Lista_argumenata) {
```

```
        // Telo konstruktora izvedene klase
```

```
}
```

*poziv konstruktora
osnovne klase*

- Kada *samo izvedena klasa* ima konstruktor, prilikom kreiranja objekta koristi se podrazumevani (*default*) konstruktor osnovne klase

Primer: Nasleđivanje klase koja *nema* definisan konstruktor (1/2)

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

class Oblik2D { // Klasa nema definisan konstruktor
public:
    double visina;
    double sirina;
    void prikaziDimenzije() {
        cout << "sirina= " << sirina << " visina= " << visina << endl;
    }
    double getVisina() const { return visina; }
    double getSirina() const { return sirina; }
    void setSirina(double s) { sirina = s; }
    void setVisina(double v) { visina = v; }
}
```

Primer: Nasleđivanje klase koja *nema* definisan konstruktor (2/2)

```
class Trougao: public Oblik2D { // Klasa ima definisan konstruktor
public:
    string vrsta;
    double povrsina() { return sirina * visina / 2; }

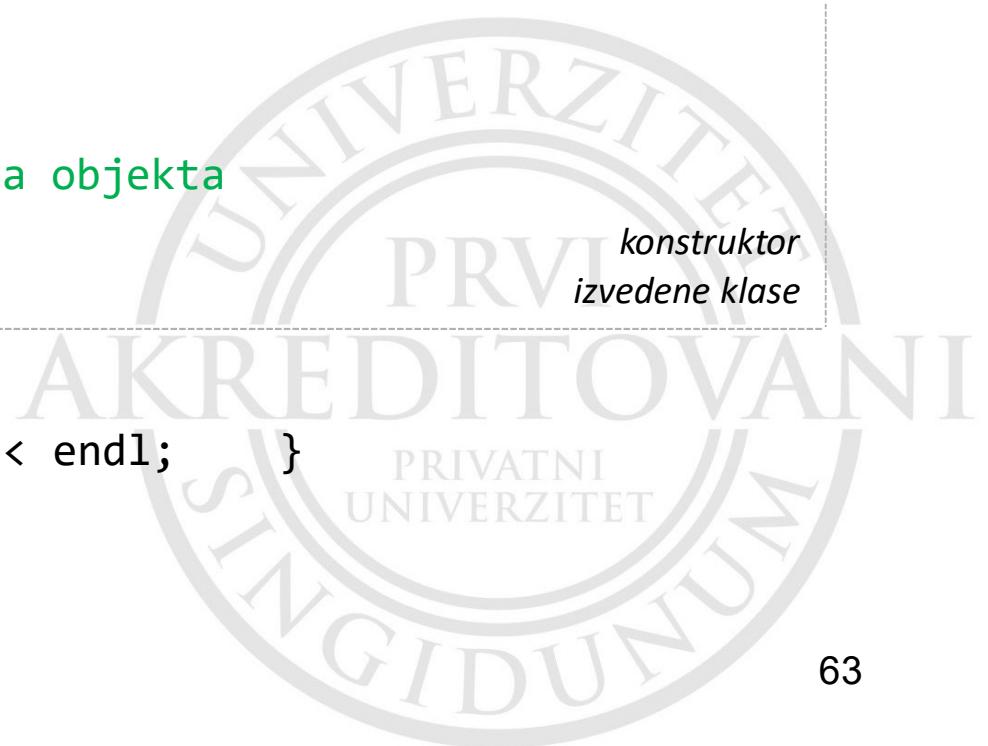
    Trougao(string tip, double v, double s) {
        // Inicijalizacija osnovnog dela objekta (kreiranog automatski)
        setSirina(s);
        setVisina(v);

        // Inicijalizacija izvedenog dela objekta
        vrsta = tip;
    }

    void prikaziVrstu() {
        cout << "Trougao je:> " << vrsta << endl;
    }
};
```

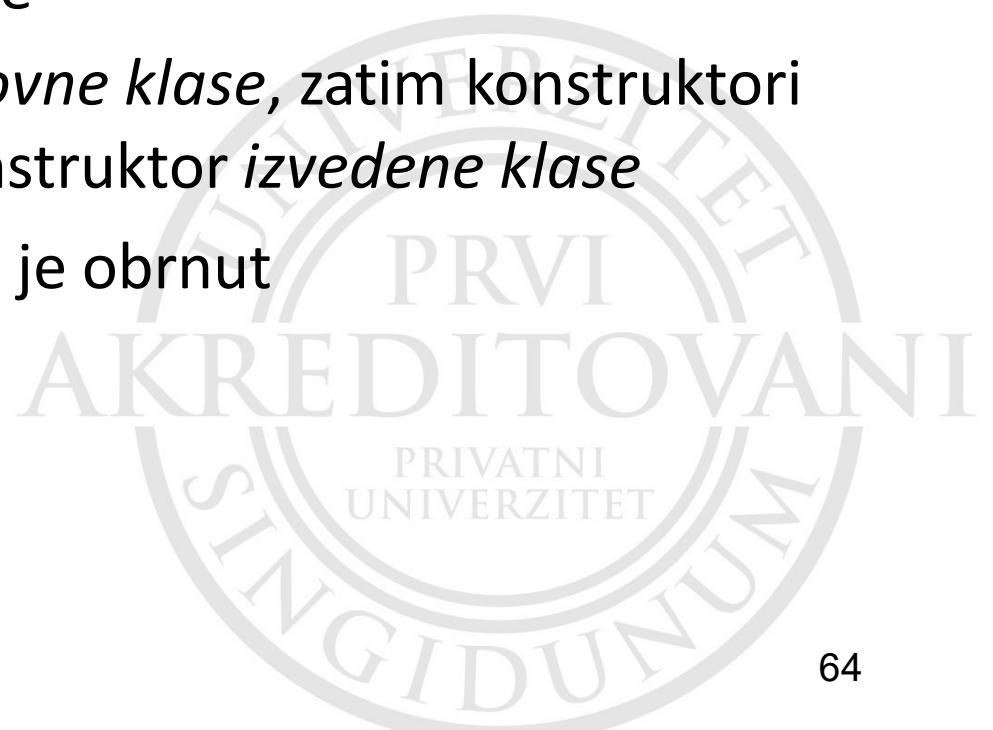
Trougao(string tip, double v, double s) {
 // Inicijalizacija osnovnog dela objekta (kreiranog automatski)
 setSirina(s);
 setVisina(v);

 // Inicijalizacija izvedenog dela objekta
 vrsta = tip;
}



Prenos argumenata konstruktoru osnovne klase

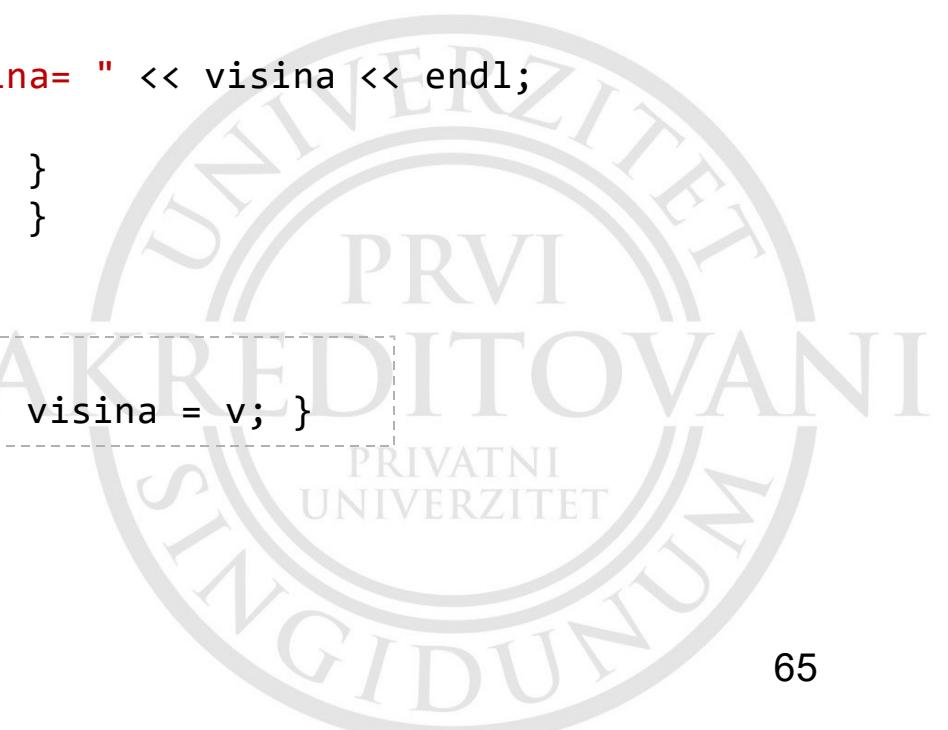
- Konstruktor osnovne klase treba da inicijalizuje deo objekta koji pripada osnovnoj klasi na osnovu *argumenata* iz poziva
- Na osnovu prenesenih aktuelnih argumenata prevodilac bira odgovarajući konstruktor *osnovne klase* i *podataka članova*, a zatim konstruktor *izvedene klase*
- Prvo se poziva konstruktor *osnovne klase*, zatim konstruktori *podataka članova* i na kraju konstruktor *izvedene klase*
- Redosled pozivanja destruktora je obrnut



Primer: Nasleđivanje klase koja ima definisan konstruktor (1/2)

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

class Oblik2D {
private:
    double visina;
    double sirina;
public:
    void prikaziDimenzije() {
        cout << "sirina= " << sirina << "  visina= " << visina << endl;
    }
    double getVisina() const { return visina; }
    double getSirina() const { return sirina; }
    void setSirina(double s) { sirina = s; }
    void setVisina(double v) { visina = v; }
    // Konstruktor osnovne klase
    Oblik2D(double s, double v) { sirina = s; visina = v; }
};
```



Primer: Nasleđivanje klase koja ima definisan konstruktor (2/2)

```
class Trougao: public Oblik2D {  
public:  
    string vrsta;  
    double povrsina() { return getSirina() * getVisina() / 2; }  
  
    Trougao(string tip, double s, double v) : Oblik2D (s,v) {  
        // Inicijalizacija izvedenog dela  
        vrsta = tip;                                konstruktor izvedene klase  
    }                                              pokreće konstruktor osnovne klase  
  
    void prikaziVrstu() {  
        cout << "Trougao je:>: " << vrsta << endl;      }  
};  
  
int main() {  
    Trougao t("jednakostranicni", 4.0, 4.0);  
    t.prikaziDimenzije();  
    t.prikaziVrstu();  
    cout << "Povrsina trougla je: " << t.povrsina() << endl;  sirina= 4 visina= 4  
    return 0;                                         Trougao je:>: jednakostranicni  
}                                                 Povrsina trougla je: 8
```

Nasleđeni konstruktori

- Izvedena klasa može da pokrene više oblika konstruktora osnovne klase, zavisno od prenesenih argumenata,npr.

```
Oblik2D::Oblik2D(double s, double v){ sirina=s; visina=v; }
Oblik2D::Oblik2D(double x){ sirina = visina = x; }
// Default konstruktor
Trougao::Trougao(){ vrsta = "nepoznat"; }
Trougao::Trougao(string tip, double v, double s) :
    Oblik2D(s, v) { vrsta = tip; }
// Konstruktor jednakostranicnog trougla
Trougao::Trougao(string tip, double x) :
    Oblik2D(x) { vrsta = "jednakostranicni"; }
```

- Ako je neki parametar konstruktora osnovne klase obavezan, nasleđene klase ga moraju proslediti

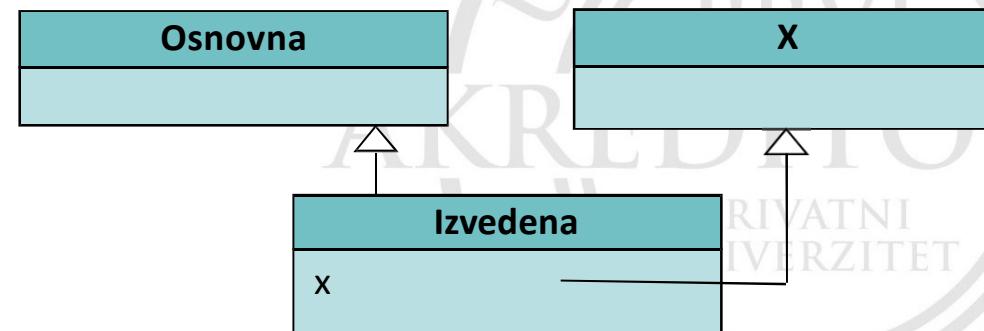
6.3 Destruktori i nasleđivanje

- Brisanje objekta nasleđene klase
- Redosled pozivanja destruktora



Brisanje objekta nasleđene klase

- Uklanjanje objekta izvedene klase pokreće izvršavanje destruktora *osnovne* i destruktora *izvedene* klase
 - redosled pozivanja destruktora je obrnut od redosleda pozivanja konstruktora, odnosno redosleda izvođenja klasa: prvo se briše *izvedena* klasa, zatim *podaci članovi* i na kraju *osnovna* klasa, čime se obezbeđuje da u memoriji ne ostaju nepotpuni objekti čijim delovima se ne može pristupiti
- Primer: Kreiranje objekta izvedene klase koja ima član podatak **x**

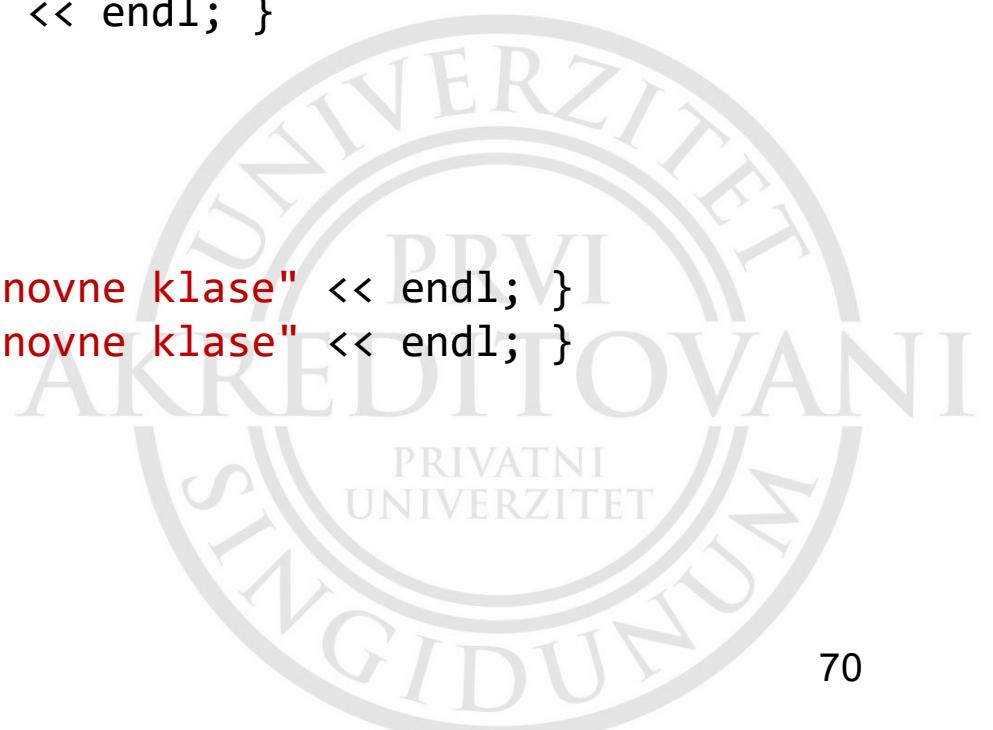


Primer: Redosled pozivanja konstruktora i destruktora kod kreiranje objekta (1/2)

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

class X {
public:
    X() { cout << "konstruktor klase X" << endl; }
    ~X() { cout << "destruktor klase X" << endl; }
};

class Osnovna {
public:
    Osnovna() { cout << "konstruktor osnovne klase" << endl; }
    ~Osnovna() { cout << "destruktor osnovne klase" << endl; }
};
```



Primer: Redosled pozivanja konstruktora i destruktora kod kreiranje objekta (2/2)

```
class Izvedena : public Osnovna {  
    X x;      // ekvivalentno "class Izvedena: private X { ... };"  
public:  
    Izvedena() { cout << "konstruktor izvedene klase" << endl; }  
    ~Izvedena() { cout << "destruktur izvedene klase" << endl; }  
};  
  
int main() {  
    // Kreiranje objekta izvedene klase (podobjekti Osnovna i X)  
    Izvedena i;  
    return 0;  
}
```

konstruktor osnovne klase
konstruktor klase X
konstruktor izvedene klase
destruktur izvedene klase
destruktur klase X
destruktur osnovne klase

6.4 Duplikati naziva članova klase

- Duplikati naziva polja podataka
- Duplikati naziva funkcija članova



Duplikati naziva polja podataka

- Nazivi *polja podataka* članova osnovne i izvedene klase mogu biti isti i mogu se koristiti pomoću operatora razrešenja dosega, npr.

```
int Izvedena::ukupno() const {  
    return vrednost + Osnovna::vrednost;  
}
```



Duplikati naziva funkcija članova

- Nazivi *funkcija članova* osnovne i izvedene klase mogu biti isti, a koriste se u zavisnosti od njihovih *parametara*
- Ako su parametri *isti*, funkcije osnovne i izvedene klase se razlikuju pomoću *naziva klase*, npr.

Izvedena objekt;
objekt.**Osnovna**::funkcija();

- Ako se parametri funkcija *razlikuju*, funkcija član izvedene klase skriva funkciju člana osnovne klase istog naziva
Tada se *u izvedenoj klasi* može definisati funkcija *novog imena* za pristup članu osnovne klase pomoću ključne reči *using*, npr.
using Osnovna::funkcija() // koristi se **osnovna**

6.5 Višestruko nasleđivanje

- Osnovne klase
- Višezačnost funkcija članova
- Ponovljeno nasleđivanje
- Virtuelne osnovne klase



Osnovne klase

- Nova klasa može istovremeno da nasledi *više osnovnih klasa*, tako što se u zaglavlju posebno navode *specifikacije* i *nazivi* svake od klasa, odvojene zarezima
- Svaki objekt izvedene klase nasleđuje *sve članove* osnovnih klasa, npr.

```
class Motocikl {  
    // prva osnovna klasa  
};  
class VoziloSTriTocka {  
    // druga osnovna klasa  
};  
class MotociklSPrikolicom: public VoziloSTriTocka, public Motocikl {  
    // izvedena klasa  
};
```



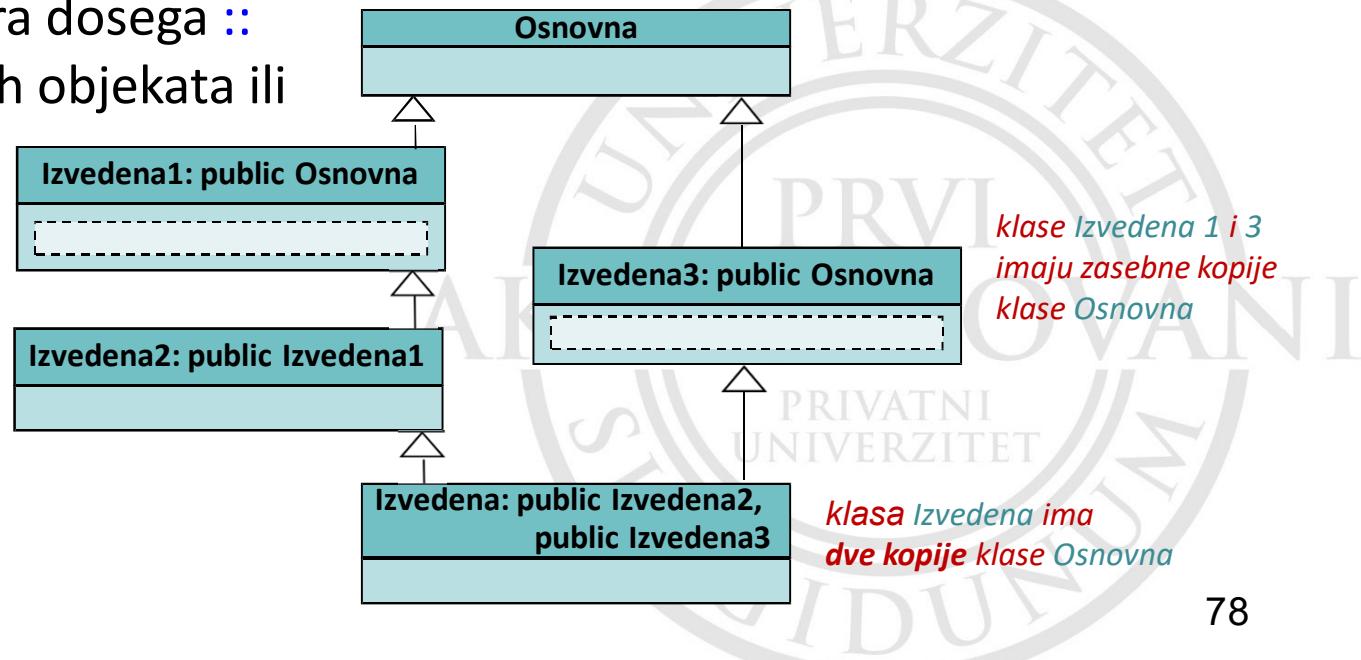
Višeznačnost funkcija članova

- Višestruko nasleđivanje može da dovede do dupliranja imena članova u različitim klasama
- Rešenja su
 1. promena naziva funkcija u klasama ili
 2. upotreba operatora dosega `::` za pune nazine svih objekata



Ponovljeno nasleđivanje

- Višestruko nasleđivanje može da dovede do pojave više verzija osnovne klase u izvedenoj klasi, *direktno* i *indirektno* nasleđenih
- Rešenja su
 - promena naziva funkcija u klasama ili
 - upotreba operatora dosega `::` za pune nazine svih objekata ili
 - upotreba *virtuelnih klasa*



Virtuelne osnovne klase

- Virtuelne klase omogućavaju izbegavanje dupliranja osnovne klase u izvedenoj klasi specifikacijom **virtual** ispred naziva osnovne klase, npr.

```
class Izvedena1: public virtual Osnovna {  
    ...  
};  
class Izvedena2: public virtual Osnovna {  
    ...  
};
```

- Na osnovu ove specifikacije prevodilac obezbeđuje da sve klase koje direktno ili indirektno nasleđuju osnovnu klasu naslede *samo jednu instancu* osnovne klase

6.6 Konverzija tipova povezanih klasa

- Svaki objekt izvedene klase ima podobjekt osnovne klase
- Konverzija objekta izvedene klase u objekte osnovne klase je automatska, tako što se uklone specifični elementi izvedene klase. Npr. objekt tipa **Kartonska kutija**

KartonskaKutija karton(30, 40, 50, "talasasti karton");

može se konvertovati u objekt tipa **Kutija**

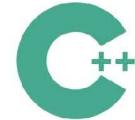
Kutija kutija;

kutija = karton;

- konverzija je moguća samo u smeru generalizacije: specifična klase se konvertuje u opšiju *odbacivanjem specifičnih elemenata* objekta, dok u obrnutom smeru konverzija nije moguća
- kod višestrukog indirektnog nasleđivanja može se pojaviti neodređenost prilikom određivanja tipa u koji objekt treba konvertovati

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. Web izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learnCPP.com/>
 - <http://www.stroustrup.com/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 05

Polimorfizam i virtuelne funkcije, preklapanje operatora u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Implementacija operatora
3. Podrazumevani članovi klase
4. Preklapanje operatora u izrazima
5. Polimorfizam i nasleđivanje
6. Virtuelne funkcije
7. Konverzije tipova



1. Uvod

- Preklapanje operatora
- Preklapanje operatora i funkcije članovi klase
- Pojam polimorfizma
- Virtuelne funkcije



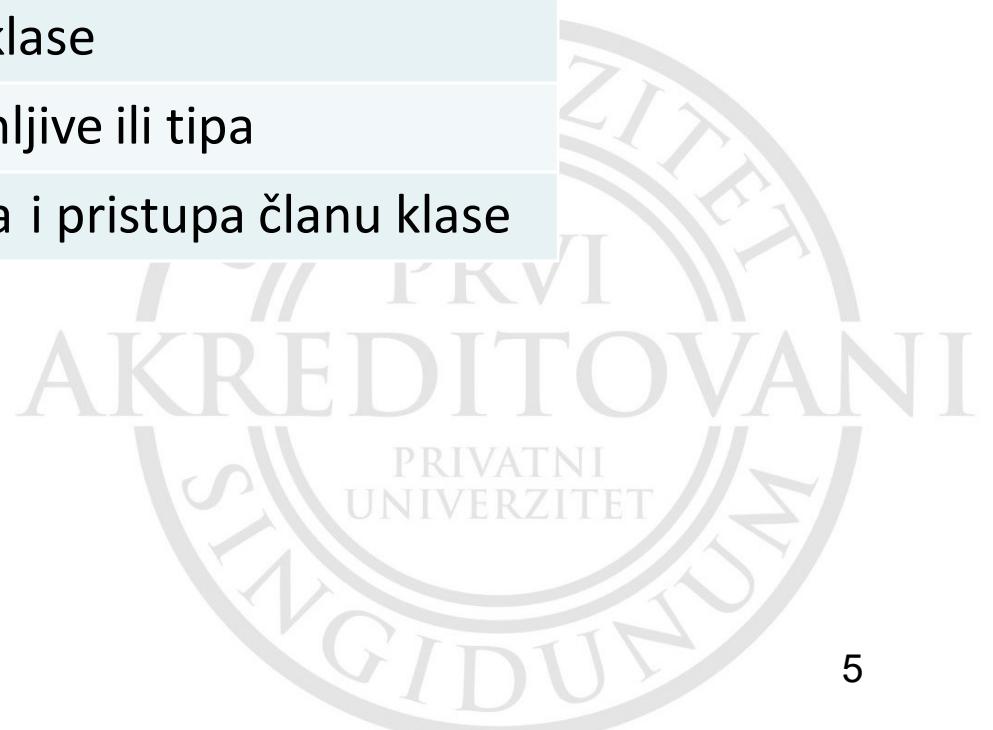
Preklapanje operatora

- Preklapanje operatora (preopterećenje, *overloading*) je svojstvo jezika C++ koje omogućava primenu standardnih operatora, kao što su `+`, `-` i `*` nad objektima novih, korisnički definisanih klasa
 - rezumljivije je `a + b * c` nego `plus(a, times(b, c))`
- Kreiraju se nove funkcije koje redefinišu operacije koje će se izvršiti nad objektima neke nove klase kada se upotrebе postojeći operatori
 - npr. da poređenje dva objekta klase *Kutija* pomoću operatora "`<`" daje rezultat `true` ako je zapremina prve kutije manja od zapremine druge
- Preklapanjem se ne mogu uvoditi novi operatori niti se može promeniti njihov prioritet prilikom evaluacije izraza

Operatori koji se ne mogu preklapati

- Neki operatori se ne mogu preklapati

Operator	Naziv
::	operator razrešenja dosega
?:	ternarni operator selekcije
.	operator pristupa članu klase
sizeof	operator veličine promenljive ili tipa
.*	operator dereferenciranja i pristupa članu klase



Preklapanje operatora i funkcije članovi klase

- Npr. za klasu **Kutija** definisanu desno, promenljive ovog tipa mogu se poređiti pomoću funkcije člana *uporedi* koja vraća vrednosti **0/1/-1** u izrazima

```
if (kutija1->uporedi(kutija2)<0)
    kutija1 = kutija2;
```

- Bilo bi jednostavnije i jasnije koristiti nove binarne operatore za poređenje promenljivih, koji vraćaju *logičku* vrednost, npr.

```
if (kutija1 < kutija2)
    kutija1 = kutija2;
```

```
class Kutija {
private:
    double duzina {1.0};
    double sirina {1.0};
    double visina {1.0};
public:
    // Konstruktori
    Kutija(double d, double s ...) ... {};
    Kutija() {}                                //default
    Kutija(const Kutija& kutija)... //k.kopije
    // Zajedno i poređenje kutija
    double zapr() const { // zapremina kutije
        return duzina*sirina*visina;
    }
    int uporedi(const Kutija& kut) {
        if (zapr() < kut.zapr()) return -1;
        if (zapr() == kut.zapr()) return 0;
        return 1;
    }
    ...
}
```

Pojam polimorfizama

- Polimorfizam je osobina objektno orijentisanih jezika koja omogućava da osnovna klasa definiše *zajedničke funkcije* izvedenih klasa, koje se mogu implementirati u izvedenim klasama na različit način
- Pošto zajednička klasa definiše *jedinstveni interfejs* izvedenih klasa, programer treba da poznaje znatno manji broj različitih interfejsa, što olakšava razvoj i održavanje složenih programa i kreiranje biblioteka funkcija
- Izvedene klase mogu implementirati sve ili samo neke od zajedničkih funkcija

Virtuelne funkcije

- U jeziku C++ polimorfizam se relizuje korišćenjem **virtuelnih funkcija**, koje se definišu pomoću deklaracije **virtual** u osnovnoj klasi
- Ponašanje virtuelnih funkcija može se promeniti u izvedenim klasama definisanjem sopstvene verzije za izvedenu klasu
- Postupak se naziva *nadjačavanje* funkcija članova (*overriding*)
 - nadjačavanje funkcija članova razlikuje se od preklapanja (*overloading*), kod koga su interfejsi preklopljenih funkcija obavezno različiti, dok su kod nadjačavanja *interfejsi funkcija isti*, samo se one nalaze u *različitim klasama*

2. Implementacija operatora

1. Preklapanje operatora klase
2. Implementacija preklopljenih operatora klase
3. Implementacija globalnih preklopljenih operatora
4. Potpuna implementacija operatora
5. Idiomi operatorskih funkcija



2.1 Preklapanje operatora klase

- Funkcija u C++ kojom se definiše novi operator ima opšti oblik
`operator Op operatorska_funkcija`
gde *Op* može biti npr. operator `+`, `--`, `new` itd.
- Operatorske funkcije mogu se definisati kao funkcije neke klase ili kao globalne funkcije
 - npr. operator poređenja "manje od" za klasu `Kutija` može se definisati

```
class Kutija {  
public:  
    bool operator < (const Kutija& kut) const; // preklapanje "<"  
    // Ostatak definicije klase ...  
}
```
 - za svaki operator je moguće definisati više preklopljenih funkcija, odnosno tumačenja nekog operatora

2.2 Implementacija preklopljenih operatora klase

- U naredbi

```
if (kutija1 < kutija2) ...
```

izraz u zagradi će pokrenuti operatorsku funkciju klase, kao da naredba ima oblik

```
if (kutija1.operator<(kutija2)) ...
```

- Pokrenuće se izvršavanje operatorske funkcije "manje od", koja poređi zapremine dva objekta klase *Kutija*

```
bool Kutija::operator<(const Kutija& kutija) const {  
    return this->zapremina() < kutija.zapremina();  
}
```

– primena *reference* kao parametra operatorske funkcije sprečava kopiranje argumenta *kutija*, kao i *const* za implicitni argument

Primer: Implementacija operatora klase Kutija (1/2)

```
# include <iostream>
using namespace std;
// Deklarisanje klase Kutija
class Kutija {
private:
    double duzina; double sirina; double visina;
public:
    // Konstruktori klase
    Kutija(double d, double s, double v) { duzina=d; sirina=s; visina=v; }
    Kutija() {} // podrazumevani (default) konstruktor
    // Pristupne funkcije i operatori klase
    double zapremina() const // funkcija računa zapreminu kutije
        { return duzina*sirina*visina; }
    double getDuzina() const { return duzina; }
    double getSirina() const { return sirina; }
    double getVisina() const { return visina; }
    bool operator<(const Kutija& kutija) const // operator "manje od"
        { return zapremina() < kutija.zapremina(); }
};
```

Primer: Implementacija operatora klase Kutija (2/2)

```
int main() {
    // Deklarisanje i inicializacija promenljivih
    Kutija kutije[4];
    kutije[0] = Kutija(2.0, 2.0, 3.0);
    kutije[1] = Kutija(1.0, 3.0, 2.0);
    kutije[2] = Kutija(1.0, 2.0, 1.0);
    kutije[3] = Kutija(2.0, 3.0, 3.0);
    // Pronalaženje najmanje kutije
    Kutija malaKutija = kutije[0];
    for (int i=1; i<=3; i++) {
        if (kutije[i] < malaKutija)
            malaKutija = kutije[i];
    }
    cout << "Najmanja kutija ima dimenzije:"
        << malaKutija.getDuzina() << "x"
        << malaKutija.getSirina() << "x"
        << malaKutija.getVisina() << endl;
}
```



2.3 Implementacija globalnih preklopljenih operatora

- Operatorska funkcija se može implementirati i izvan klase, kao *globalna* operatorska funkcija
- Parametar lista tada sadrži potpuni broj parametara, npr. funkcija "manje od" klase **Kutija** ima dva parametra

```
inline bool operator<(const Kutija& kutija1, const Kutija&
    kutija2) {
    return kutija1.zapremina() < kutija2.zapremina();
}
```

- operatorska funkcija je *inline* da bi se prevodila u svakom fajlu u kome se upotrebi
- specifikacija **const** se u globalnoj operatorskoj funkciji ne navodi, jer se primenjuje samo na *funkcije članove klase*, kao oznaka da funkcija ne menja objekt (implicitni parametar)

2.4 Potpuna implementacija operatora

- Implementacija operatora klase treba da predviđi sve *različite načine upotrebe*, npr. različite tipove argumenata
- Tako se operator "manje od" klase **Kutija** može proširiti tako da dozvoljava izraze kao **kutija>5.0** ili **10.0<kutija**

```
// Poređenje zapremine kutije s konstantom
inline bool Kutija::operator<(double k_vred) const {
    return zapremina() < k_vred;
}
// Poređenje konstante sa zapreminom kutije
inline bool operator<(double k_vred, const Kutija& kutija) {
    return k_vred < kutija.zapremina();
}
```

2.5 Forme operatorskih funkcija

- Prilikom implementacije operatorskih funkcija mora se definisati *tačan broj* parametara (najčešće 1 ili 2)
- Binarni preklopljeni operator klase je funkcija član za koju je levi operand iste klase

Tip_vrednosti operator Op(Tip_op desni_operand);

- tip vrednosti zavisi od vrste operatora, npr. za relacione i logičke operatore može biti logička vrednost

- Binarni operator koji *nije* član klase može se definisati kao

*Tip_vrednosti operator Op (Tip_klase Levi_operand,
Tip_op desni_operand);*

- levi operand je iz klase za koju je definisano preklapanje, a desni operand može biti bilo kog tipa, uključujući i *Tip_klase*

Forme operatorskih funkcija

- **Unarni** operatori se mogu implementirati kao funkcije bez parametara ako su članovi klase

```
Tip_klase& operator Op();
```

- Ako operatorske funkcije nisu članovi klase, implementiraju se kao *globalne* operatorske funkcije s jednim parametrom

```
Tip_klase& operator Op(Tip_klase& operand);
```



3. Podrazumevani članovi klase

1. Vrste podrazumevanih funkcija članova klase
2. Definisanje destruktora
3. Definisanje konstruktora kopije
4. Definisanje podrazumevajućeg operatora dodele vrednosti



3.1 Vrste podrazumevanih funkcija članova klase

- Prevodilac može da *automatski* kreira podrazumevani konstruktor i konstruktor kopije; npr. za jednostavnu klasu koja ima samo jedno polje podataka

```
class Podaci {  
    public:  
        int vrednost;  
}
```

prevodilac kreira sledećih 6 podrazumevanih članova klase:

Podaci();	// podrazumevani konstruktor
Podaci(const Podaci& podatak);	// konstruktor kopije
Podaci(Podaci&& podatak);	// konstruktor premeštanja
~Podaci();	// destruktur
Podaci& operator =(const Podaci& podatak);	// operator dodele
Podaci& operator =(const Podaci&& podatak);	// op. dodele premešt.

Upotreba podrazumevanih funkcija članova klase

- Problemi s automatski kreiranim funkcijama članovima
 - podrazumevani *konstruktor kopije* vrši jednostavno kopiranje podataka, čak i pokazivača, tako da se stvaraju *kopije pokazivača* na iste objekte
 - podrazumevani operator dodele vrši takođe jednostavno kopiranje podataka, uključujući pokazivače
 - podrazumevani *operator premeštanja*, kao i *operator dodele i premeštanja* na isti način koriste desnu stranu u obliku reference
- Rešenje: ako će se koristiti automatske podrazumevane funkcije, navodi se ključna reč *default*, inače *delete*:

```
Podaci() = default;
```

// koristi se

```
Podaci(const Podaci& podaci)=delete; // ne koristi se
```

```
Podaci& operator=(const Podaci& podaci)=delete; // net koristi
```

3.2 Definisanje destruktora

- Destruktor klase treba da oslobodi memoriju (vrati u *heap*)
- Ako je privatna promenljiva ptext član klase koji predstavlja *pokazivač* definisan kao `string* ptext`, a konstruktorska funkcija alocira memoriju naredbom

```
ptext = new string {tekst};
```

destruktor oslobađa memoriju naredbom

```
delete ptext;
```

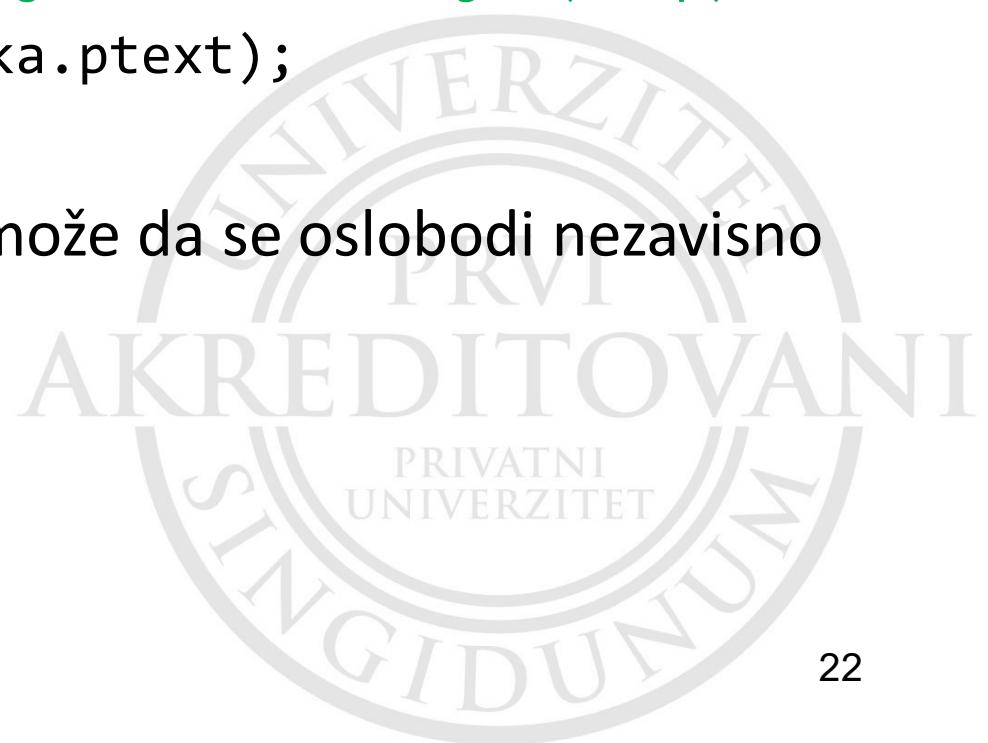
- To ipak nije dovoljno, jer nastaje problem kad se tekst prenosi po vrednosti, pa se pokrene *konstruktor kopije* i proizvede *kopiju pokazivača*, koja pokazuje na istu memoriju

3.3 Definisanje konstruktora kopije

- Kada klasa koristi članove podatke koji su *pokazivači*, konstruktor kopije, umesto kopije pokazivača, treba da kreira *novi objekt*, koji ne zavisi od originalnog, npr.

```
Poruka(const Poruka& poruka) {  
    // Kreiranje duplikata objekta u memoriji (heap)  
    ptext = new string(*poruka.ptext);  
}
```

- Memorija originalnog objekta može da se oslobodi nezavisno od novog objekta



3.4 Definisanje podrazumevanih operatora dodele vrednosti

- Operator dodele vrednosti za promenljivu `string* ptext`

```
Poruka& operator=(const Poruka& poruka) {  
    ptext = new string(*poruka.ptext); // duplikat objekta  
    return *this;  
}
```

može se uobičajeno koristiti u naredbama kao što je, npr.

```
poruka1 = poruka2 = poruka3;
```

- Poseban slučaj je naredba oblika

```
poruka1 = poruka1;
```

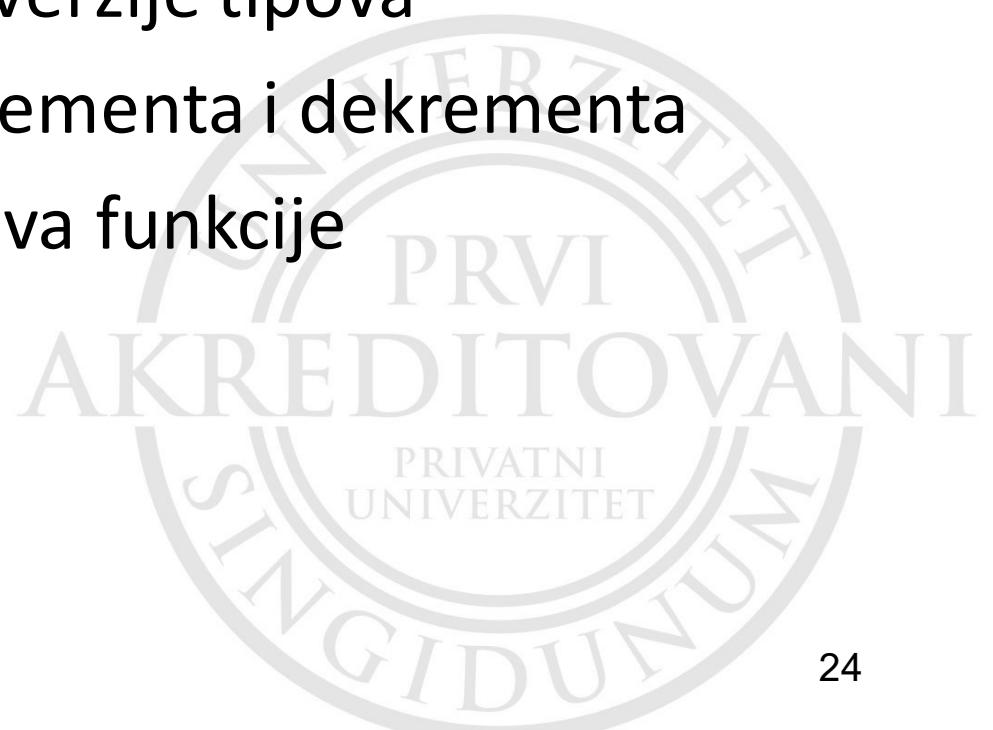
kada operator dodele kreira još jednu kopiju

- Zato je potrebno u konstruktoru proveriti ovakav slučaj

```
if (this == &poruka) return *this //samo vraća levi operand
```

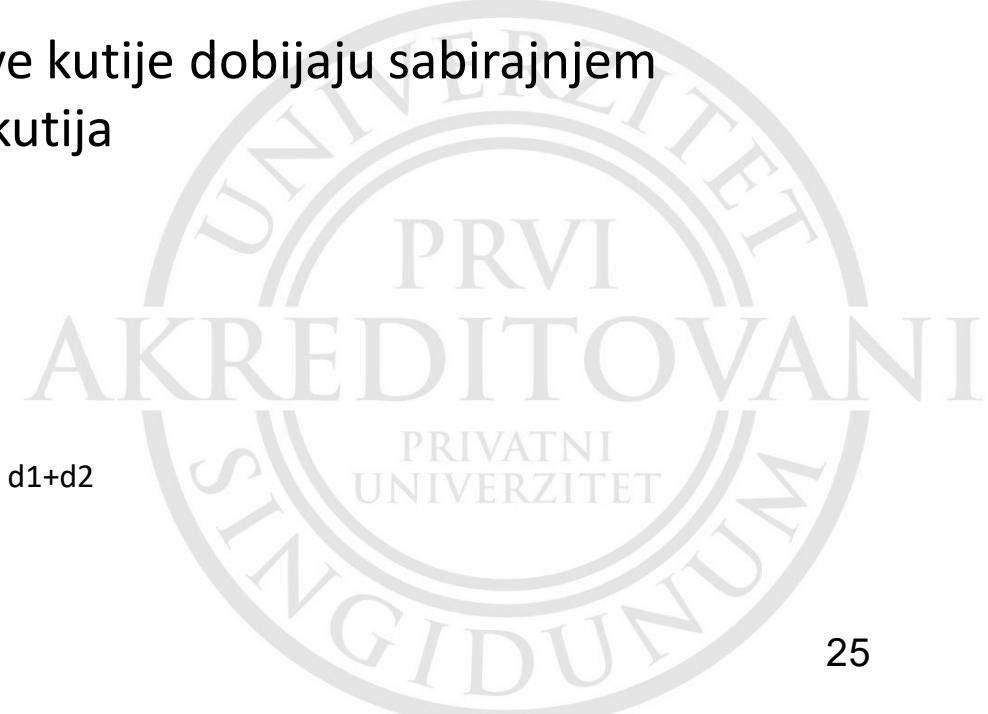
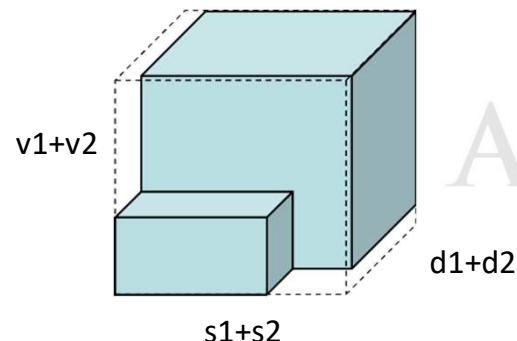
4. Preklapanje operatora u izrazima

1. Preklapanje aritmetičkih operatora
2. Preklapanje operatora poređenja
3. Preklapanje operatora indeksiranja
4. Preklapanje operatora konverzije tipova
5. Preklapanje operatora inkrementa i dekrementa
6. Preklapanje operatora poziva funkcije



4.1 Preklapanje aritmetičkih operatora

- Preklapanje aritmetičkih operacija omogućava pregledno pisanje operacija nad novim klasama objekata
- Primer je definicija sabiranja (+) za objekte klase **Kutija**, gde se zbir dva objekta može definisati kao nova kutija, u koju one mogu istovremeno da stanu
 - to znači da se sve tri dimenzije nove kutije dobijaju sabirajnjem odgovarajućih dimenzija polaznih kutija



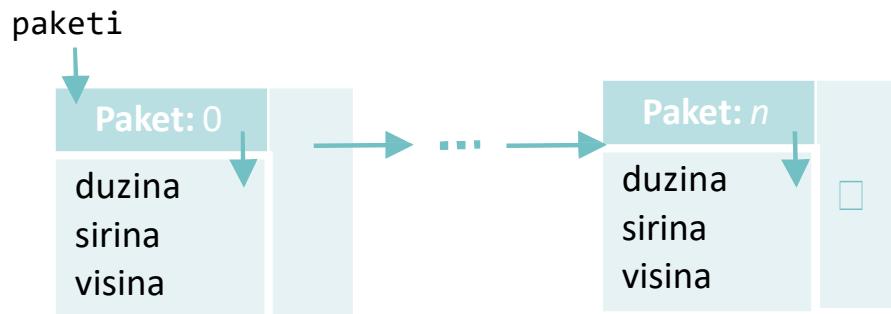
4.2 Preklapanje operatora poređenja

- Preklapanje operatora poređenja omogućava pregledno pisanje logičkih izraza u kojima se porede objekti novih, korisnički definisanih klasa objekata
 - npr. poređenje različitih objekata klase **Kutija** na osnovu njihove zapreme (prethodni primeri)



4.3 Preklapanje operatora indeksiranja

- Omogućava pristup elementima različitih struktura, kao što su retko zaposednute matrice, asocijativna polja ili povezane liste na način kako se pristupa elementima polja
 - npr. Paket[1] može da predstavlja element strukture povezane liste objekata tipa **Kutija**, a ne polje
- Preklapanje omogućava definisanje različitih internih operacija pronalaženja elementa određenog tipa u ovakvim strukturama



```
class Paket {  
    private:  
        Kutija* pKutija; // pokaz. na obj. Kutija  
        Paket* pSled; // pokaz. na sled. paket  
    public:  
        ...  
};
```

Primer: Pronalaženje objekta pomoću operatora indeksiranja

- Pronalaženje odgovarajućeg objekta klase **Kutija** u strukturi paketi klase **Paket** može se izvršiti pomoću preklopljenog operatora indeksiranja "[]":

```
inline Kutija* operator[](unsigned int index) const {  
    Paket* p = paketi;      // pokazivač na prvi paket u listi  
    unsigned int count = 0; // brojač paketa  
    do {  
        if (index == count++) // da li je pronađen indeks paketa  
            return p->pKutija; // ako jeste, pokazivač na Kutija  
    } while (p = p->pSled); // sve dok postoji sledeći != Null  
    return nullptr;  
}
```

4.4 Preklapanje operatora konverzije tipova

- Preklapanje operatora može se koristiti za konverzije tipova objekata u neki od standardnih tipova ili tip klase, a vrši se definisanjem operatorske funkcije *Tip()*
- Npr. konverzija objekta tipa **Kutija** u objekt tipa **double**:

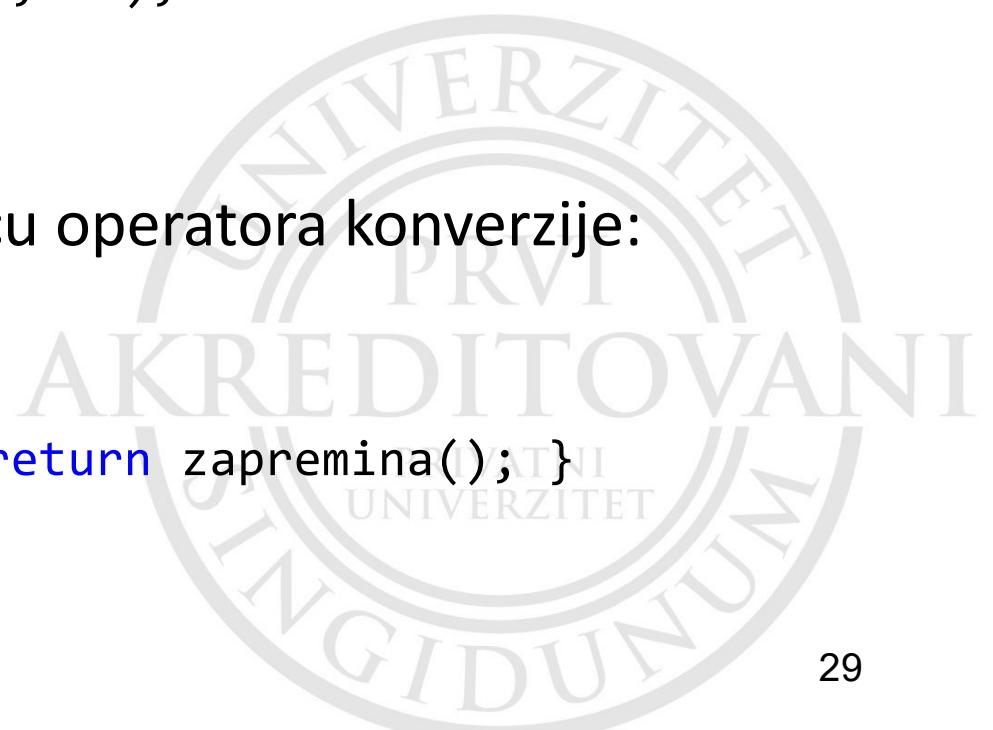
```
Kutija kutija = Kutija(1.0,2.0,3.0);
```

```
double zapreminaKutije;
```

```
zapreminaKutije = kutija;
```

može se vršiti *implicitno* pomoću operatora konverzije:

```
class Kutija {  
public:  
    operator double() const { return zapremina();}  
    ...  
};
```



4.5 Preklapanje operatora inkrementa i dekrementa

- Specifičnost ovih operatora je da se u izrazima mogu koristiti *prefiksno* i *postfiksno*. Za svaku varijantu definiše se posebna operatorska funkcija, npr.

```
class Object {  
public:  
    ...  
    Object& operator++();           // prefiksni inkrement  
    ...  
    const Object operator++(int);   // postfiksni inkrement  
    ...  
};
```

- Operatorske funkcije vraćaju referencu `*this` na tekući izmenjeni objekt; `int` argument služi prevodiocu da ih razlikuje

4.5 Preklapanje operatora poziva funkcije

- Poziv objekta tipa *funkcije* (tzv. funktora) takođe se može preklopiti definisanjem operatorske funkcije "()", npr.

```
class Zapremina {  
public:  
    double operator()(double x, double y, double z) {  
        return x*y*z  
    }  
    ...  
}
```

koja se poziva kao

```
Zapremina zapremina;  
double soba = zapremina(16,12,8.5) // zapremina sobe u m3
```



5. Polimorfizam i nasleđivanje

1. Pokazivači na objekte izvedenih klasa
2. Nadjačavanje funkcija u izvedenim klasama



5.1 Pokazivači na objekte izvedenih klasa

- U jeziku C++ pokazivači se mogu dodeljivati samo promenljivima istog osnovnog tipa, izuzev pokazivača na objekte *osnovne klase* i klasa *izvedenih* iz osnovne klase, npr.

```
Osnovna osnovni_obj;  
Izvedena izvedeni_obj;  
Osnovna *p;  
// Ispravne obe naredbe  
p = &osnovni_obj;  
p = &izvedeni_obj;
```

- pokazivač na objekt *osnovne klase* može se koristiti za pristup samo onim delovima objekta izvedene klase koji su nasleđeni
- pokazivačima na objekte *izvedene klase* ne može se dodeliti vrednost pokazivača na objekt osnovne klase

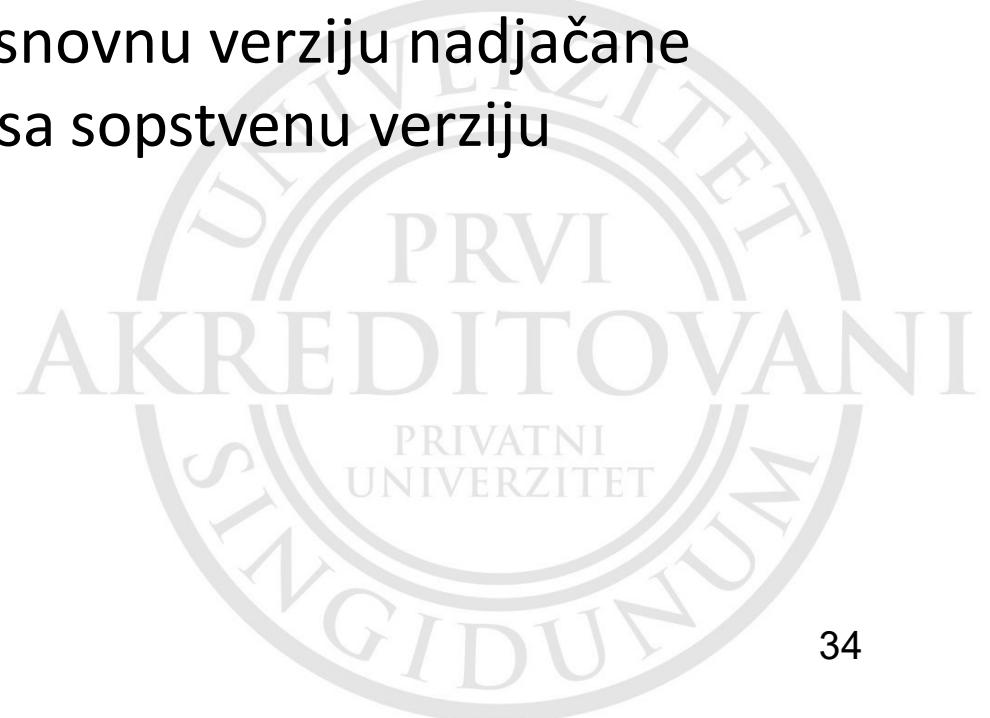
5.2 Nadjačavanje funkcija u izvedenim klasama

- Nadjačavanje funkcija omogućava nasleđenoj klasi da definiše sopstvenu implementaciju zajedničke funkcije, npr.

```
double Oblik2D::povrsina(){return getSirina()*getVisina();}  
double Trougao::povrsina(){return getSirina()*getVisina()/2;}
```

– nadjačane funkcije imaju *isti* tip rezultata, broj i tip parametara

- Objekti osnovne klase koriste osnovnu verziju nadjačane funkcije, a objekti izvedenih klasa sopstvenu verziju



Primer: Upotreba nadjačane funkcije

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

class Osnovna {
public:
    void prikaziPoruku() {
        cout << "Osnovna klasa" << endl;
    }
};

class Izvedena : public Osnovna {
public:
    void prikaziPoruku() {
        cout << "Izvedena klasa" << endl;
    }
};

int main () {
    Osnovna osnovni; Izvedena izvedeni;
    osnovni.prikaziPoruku(); izvedeni.prikaziPoruku(); // klase pokreću svoju verz.
    return 0;
}
```

Osnovna klasa
Izvedena klasa

6. Virtuelne funkcije

1. Definisanje virtuelne funkcije
2. Podrazumevane vrednosti argumenata virtuelne funkcije
3. Pozivanje virtuelnih funkcija
4. Pozivanje verzije virtuelne funkcije iz osnovne klase
5. Čiste virtuelne funkcije
6. Virtuelni destruktori



6.1 Definisanje virtuelne funkcije

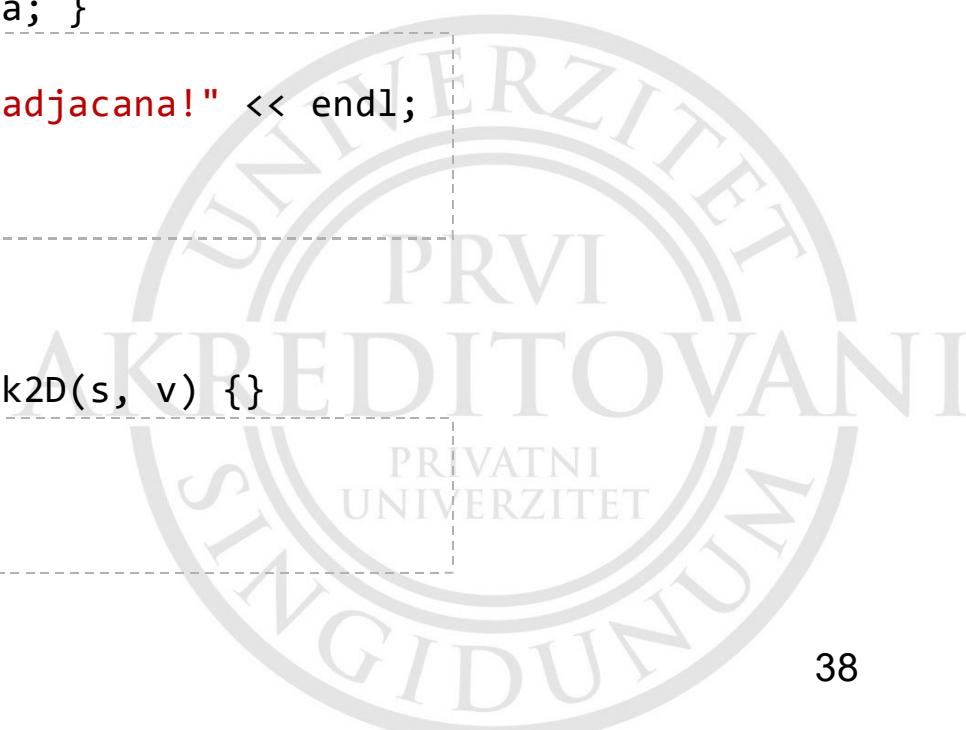
- Funkcija se deklariše kao virtuelna u osnovnoj klasi pomoću deklaracije

```
virtual tip_vrednosti naziv_funkcije (lista_parametara) {  
    // telo funkcije  
}
```

- Klasa koja sadrži virtuelnu funkciju naziva se *polimorfna klasa*
 - funkcija deklarisana kao virtuelna ostaje takva u svim nasleđenim klasama
- Prilikom pozivanja funkcije pomoću pokazivača, u toku izvršavanja se određuje verzija nadjačane funkcije koja će se pokrenuti na osnovu stvarnog tipa objekta
- *Pokazivači su neophodni za realizaciju polimorfizma*

Primer: Definisanje i upotreba nadjačane funkcije (1/2)

```
# include <iostream>
using namespace std;
class Oblik2D {
    double sirina;
    double visina;
public:
    Oblik2D(double s, double v) { visina = v; sirina = s; }
    double getVisina() const { return visina; }
    double getSirina() const { return sirina; }
    virtual double povrsina() {
        cout << "Greska! funkcija mora biti nadjacana!" << endl;
        return 0.0;
    }
};
class Pravougaonik : public Oblik2D {
public:
    Pravougaonik(double s, double v) : Oblik2D(s, v) {}
    double povrsina() {
        return getVisina()*getSirina();
    }
};
```



Primer: Definisanje i upotreba nadjačane funkcije (2/2)

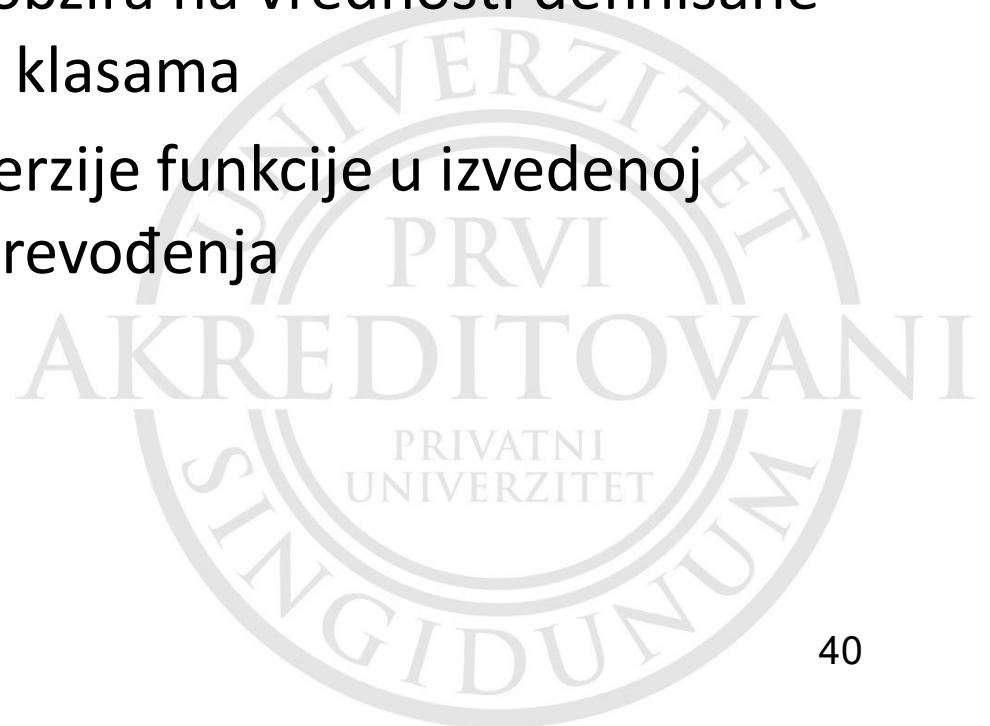
```
class Trougao: public Oblik2D {  
public:  
    Trougao(double s, double v) : Oblik2D(s, v) {}  
    double povrsina() {  
        return getVisina()*getSirina()/2;  
    }  
};  
  
int main () {  
    Oblik2D oblik2D(3, 4);           // objekt osnovne klase  
    Trougao trougao(3, 4);          // objekt klase trougao  
    Pravougaonik pravougaonik(3, 4); // objekt klase pravougaonik  
    Oblik2D *poblik;  
    poblik = &oblik2D;  
    cout << poblik->povrsina() << endl; // funkcija osnovne klase  
    poblik = &trougao;  
    cout << poblik->povrsina() << endl; // klase trougao  
    poblik = &pravougaonik;  
    cout << poblik->povrsina() << endl; // klase pravougaonik  
    return 0;  
}
```

Greska! funkcija mora biti nadjacana!

0
6
12

6.2 Podrazumevane vrednosti argumenata virtuelne funkcije

- Podrazumevane vrednosti parametara funkcije koriste se prilikom prevodenja funkcije
- Za *virtuelne funkcije* definisanje podrazumevanih vrednosti nema mnogo smisla, jer se uvek koriste vrednosti koje su definisane u osnovnoj klasi, bez obzira na vrednosti definisane u verzijama funkcije u izvedenim klasama
- Izuzetak je samo direktni poziv verzije funkcije u izvedenoj klasi, pošto se razrešava u toku prevodenja

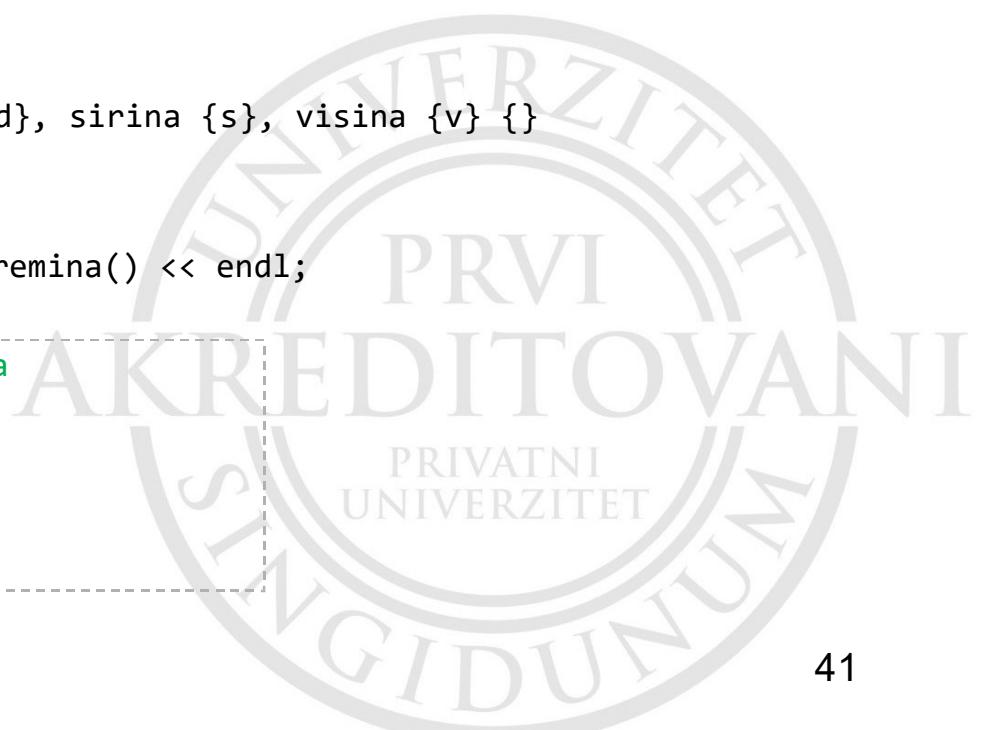


Primer: Podrazumevani parametri virtuelne funkcije (1/4)

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

class Kutija {
protected:
    double duzina = 1.0;
    double sirina = 1.0;
    double visina = 1.0;

public:
    Kutija(double dv, double s, double v) : duzina {d}, sirina {s}, visina {v} {}
    // Funkcija za prikaz zapremljene objekta
    void prikaziZapreminu() const {
        cout << "Kutija ima korisnu zapreminu " << zapremina() << endl;
    }
    // Funkcija koja računa zapreminu objekta klase Kutija
    virtual double zapremina(int i=5) const {
        cout << "Kutija za parametar = " << i << endl;
        return duzina*sirina*visina;
    }
};
```



Primer: Podrazumevani parametri virtuelne funkcije (2/4)

```
class KartonskaKutija : public Kutija {  
private:  
    string materijal;  
public:  
    KartonskaKutija(double d, double s, double v, string str = "materijal") : Kutija {d, s, v} {  
        materijal = str;  
    }  
    double zapremina(int i = 50) const override { // zapremina objekta klase KartonskaKutija  
        cout << "KartonskaKutija za parametar = " << i << endl;  
        double zap = (duzina - 0.5)*(sirina - 0.5)*(visina - 0.5);  
        return zap > 0.0 ? zap : 0.0;  
    }  
};  
class TvrdaKutija : public Kutija {  
public:  
    TvrdaKutija(double d, double s, double v) : Kutija {d, s, v} {}  
    double zapremina(int i = 500) const override { // zapremina objekta klase TvrdaKutija (-15%)  
        cout << "TvrdaKutija za parametar = " << i << endl;  
        return 0.85*duzina*sirina*visina;  
    }  
};
```

Primer: Podrazumevani parametri virtuelne funkcije (3/4)

```
int main() {
    Kutija kutija {20.0, 30.0, 40.0};                                // kutija
    TvrdaKutija jakaKutija {20.0, 30.0, 40.0};                      // izvedena kutija
    KartonskaKutija kartonskaKutija {20.0, 30.0, 40.0, "plastika"}; // izvedena kutija
    kutija.prikaziZapreminu();                                     // zapremina objekta Kutija
    jakaKutija.prikaziZapreminu();                                   // zapremina objekta TvrdaKutija
    kartonskaKutija.prikaziZapreminu();                             // zapremina objekta KartonskaKutija
    cout << "jakaKutija ima zapreminu " << jakaKutija.zapremina() << endl; // direktni poziv
    // Upotreba pokazivača na osnovnu klasu ...
    Kutija* pKutija = &kutija;                                       // pokazivač na tip Kutija
    cout << "\nZapremina objekta Kutija preko pKutija je " << pKutija->zapremina() << endl;
    pKutija->prikaziZapreminu();
    pKutija = &jakaKutija;                                         // pokazivač na tip TvrdaKutija
    cout << "Zapremina objekta jakaKutija preko pKutija je " << pKutija->zapremina() << endl;
    pKutija->prikaziZapreminu();
    pKutija = &kartonskaKutija;                                    // pokazivač na tip KartonskaKutija
    cout << "Zapremina objekta kartonskaKutija preko pKutija je " << pKutija->zapremina() <<
        endl;
    pKutija->prikaziZapreminu();
    return 0;
}
```

Primer: Podrazumevani parametri virtuelne funkcije (4/4)

```
Kutija za parametar = 5
Kutija ima korisnu zapreminu 24000
TvrdaKutija za parametar = 5
Kutija ima korisnu zapreminu 20400
KartonskaKutija za parametar = 5
Kutija ima korisnu zapreminu 22722.4
TvrdaKutija za parametar = 500
jakaKutija ima zapreminu 20400
Kutija za parametar = 5

Zapremina objekta Kutija preko pKutija je 24000
Kutija za parametar = 5
Kutija ima korisnu zapreminu 24000
TvrdaKutija za parametar = 5
Zapremina objekta jakaKutija preko pKutija je 20400
TvrdaKutija za parametar = 5
Kutija ima korisnu zapreminu 20400
KartonskaKutija za parametar = 5
Zapremina objekta kartonskaKutija preko pKutija je 22722.4
KartonskaKutija za parametar = 5
Kutija ima korisnu zapreminu 22722.4
```

Direktan poziv funkcije u izvedenoj klasi.

vrednost parametra *virtuelne*,
već verzije funkcije u izvedenoj klasi



6.3 Pozivanje virtuelnih funkcija

- Funkcija koja će se pokrenuti može se odrediti:
 - **statički**, u toku prevođenja (rano povezivanje, *early binding*)
 - **dinamički**, u toku izvršavanja (kasno povezivanje, *dynamic binding*)
- Statički se povezuju bibliotečne i preklopljne funkcije, jer su sve nepodne informacije poznate u toku prevođenja
- Dinamički se povezuju **virtuelne** funkcije, jer se tek na osnovu stvarnog tipa objekta može odrediti koju verziju funkcije je potrebno pokrenuti
- Dinamičko povezivanje je fleksibilnije, ali i *sporije* od statičkog načina povezivanja

6.4 Pozivanje verzije virtuelne funkcije iz osnovne klase

- Verzija funkcije u izvedenoj klasi može se pozvati putem pokazivača ili reference na objekt izvedene klase
- Ako je ipak potrebno pozvati funkciju *osnovne* klase za objekt *izvedene* klase može se upotrebiti funkcija *static_cast* za konverziju objekta izvedene klase u objekt osnovne klase i pokrene izvršavanje funkcije
- Npr. gubitak zapremine objekta klase *KartonskaKutija* može se izračunati kao

```
double razlikaZapremine =  
    static_cast<Kutija>(kartonskaKutija).zapremina() -  
    kartonskaKutija.zapremina();
```
- Oba poziva se razrešavaju statički, prilikom prevodenja

6.5 Čiste virtuelne funkcije

- Osnovna klasa ponekad definiše samo opšti oblik izvedenih klasa i nema nikakvu implementaciju virtuelne funkcije, kao npr. funkcija `povrsina()` klase `Oblik2D`
- U jeziku C++ takve funkcije koje nemaju implementaciju u osnovnoj klasi i moraju da budu nadjačane nazivaju se *čiste virtuelne funkcije* i deklarišu se kao

```
virtual tip naziv_funkcije (Lista_parametara) {} = 0;
```
- Svaka od izvedenih klasa *mora* da definiše svoju implementaciju čiste virtuelne funkcije
- Klasa koja ima bar jednu čistu virtuelnu funkciju zove se *apstraktna klasa*. Ne postoje objekti apstraktnih klasa, ali postoje pokazivači na ove objekte

6.6 Virtuelni destruktori

- Ne postoje konstruktori virtuelnih klasa, ali postoje *virtuelni destruktori*, koji uklanjaju objekte izvedenih klasa, jer je njihov stvarni tip nakon kreiranja poznat (dinamičko povezivanje)
- Definiciji destruktora dodaje se specifikacija **virtual**



Primer: Upotreba virtuelnog destruktora

```
# include <iostream>
using namespace std;
class Osnovna {
public:
    virtual ~Osnovna() {
        cout << "Virtuelni destruktor osnovne klase" << endl;
    }
};
class Izvedena : public Osnovna{
public:
    virtual ~Izvedena() {
        cout << "Virtuelni destruktor izvedene klase" << endl;
    }
};
void osloboodi(Osnovna *p) {
    delete p; // pokreće se virtuelni destruktor
}
int main() {
    Osnovna *posn = new Osnovna;    Izvedena *pizv = new Izvedena;
    osloboodi(posn); osloboodi(pizv);
    return 0;
}
```

Virtuelni destruktor osnovne klase
Virtuelni destruktor izvedene klase
Virtuelni destruktor osnovne klase

7. Konverzije tipova

- Konverzija između pokazivača i objekata klase
- Dinamička konverzija
- Konverzija referenci
- Ustanovljavanje polimofnog tipa



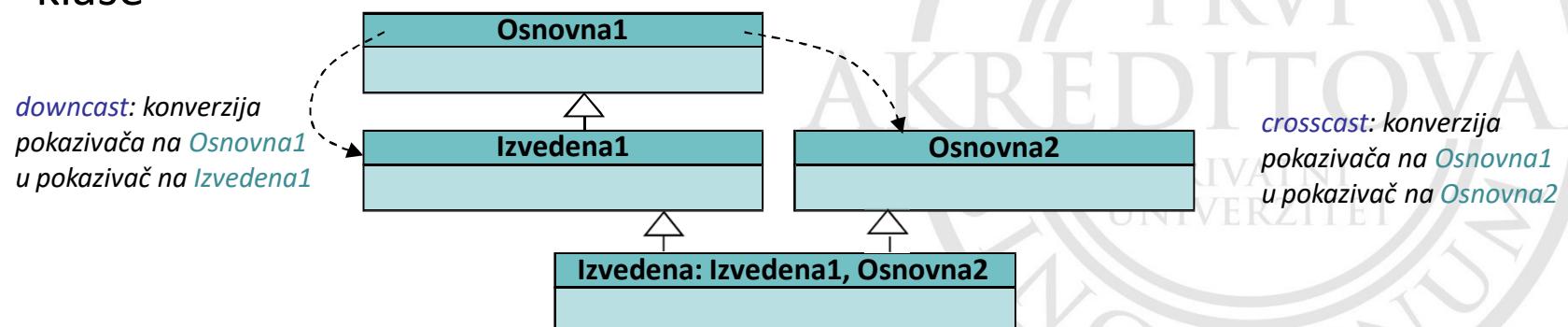
Konverzija između pokazivača i objekata klase

- Moguće je izvršiti *implicitnu konverziju* pokazivača na izvedenu klasu u pokazivač na osnovnu klasu, kako za direktno, tako i za indirektno nasleđene klase
- Np. pokazivač na klasu **KartonskaKutija** može se implicitno konvertovati:

```
KartonskaKutija* pKartonskaKutija = new
KartonskaKutija{30,40,10};
Kutija* pKutija = pKartonskaKutija;
```
- Rezultat je pokazivač na klasu **Kutija**, koji se inicijalizuje da pokazuje na *novi objekt* klase **KartonskaKutija**

Dinamička konverzija

- Dinamička konverzija vrši se u toku izvršavanja programa
 - operator `dynamic_cast<>` primenjuje se na pokazivače i reference polimorfnih tipova klasa koje sadrže najmanje jednu virtualnu funkciju
- Postoje dva oblika dinamičke konverzije tipova:
 - konverzija *niz* hijerarhiju klasa (*cast down a hierarchy*), od pokazivača na osnovnu klasu prema izvedenoj klasi
 - konverzija *kroz* hijerarhiju klasa (*cast across a hierarchy*), od pokazivača na osnovnu klasu prema drugoj osnovnoj klasi višestruko izvedene klase



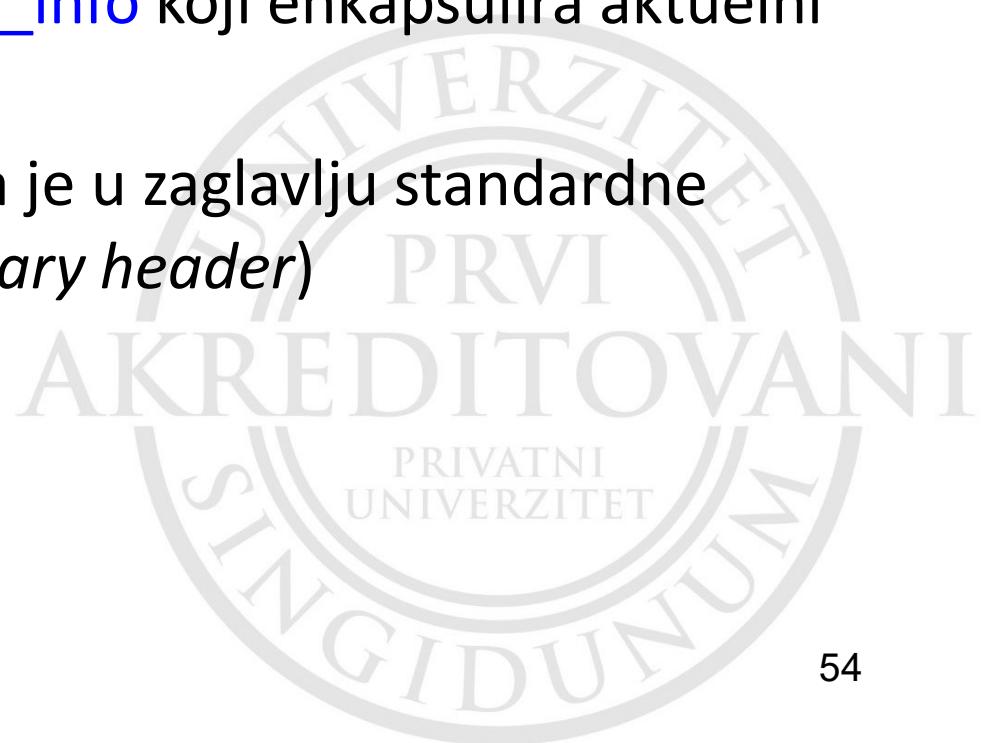
Konverzija referenci

- Moguće je primeniti operator `dynamic_cast<>()` na referencu koja je parametar funkcije radi konverzije niz hijerarhiju klasa kojom se proizvode druge reference
- Npr. parametar `rKutija` funkcije `Funkcija` je referenca na objekt osnovne klase `Kutija`. U telu funkcije se može konvertovati u referencu na izvedeni tip:

```
double Funkcija(Kutija& rKutija) {  
    ...  
    KartKutija& karton = dynamic_cast<KartKutija&>(rKutija);  
    ...  
}
```

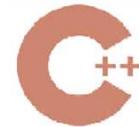
Ustanovljavanje polimofnog tipa

- Primena operatora dinamičke konverzije referenci može biti rizična ako se ne provere tipovi pokazivača ili referenci pomoću operatora `typeid()`
- Operator se može primeniti na tip ili izraz
- Operator vraća objekt `std::type_info` koji enkapsulira aktuelni tip
- Klasa `std::type_info` deklarisana je u zaglavlu standardne biblioteke rutina (*Standard Library header*)



Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Horton I., *Beginning C++*, Apress, 2014
5. Horton I., *Beginning Visual C++ 2013*, Wox/John Wiley&Sons, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. Web izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learnCPP.com/>
 - <http://www.stroustrup.com/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*



Tema 07

Upravljanje izuzecima u jeziku C++

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2



Sadržaj

1. Uvod
2. Rukovanje izuzecima
3. Objekti klase kao izuzeci
4. Funkcije koje prijavljuju izuzetke
5. Standardna biblioteka izuzetaka
6. Primeri



1. Uvod

- Obrada grešaka u programu
- Upotreba izuzetaka



Obrada grešaka u programu

- Programi se pišu tako da se predvide različite varijante njihovog izvršavanja, ustanove najčešće greške u njihovom radu i isprave se na mestu otkrivanja
 - primeri grešaka su pogrešno uneti podaci, nedozvoljene vrednosti argumenata funkcija i različite greške u računanju vrednosti izraza, npr.

```
double deljenje(float brojilac, float imenilac) {
    if (imenilac == 0) {
        cout << "GREŠKA: deljenje s nulom nije definisano!\n";
        return 0 ← dvosmislen rezultat
    }
    else
        return static_cast<double> brojilac / imenilac;
}
```

- bolji *tradicionalni* način rukovanja greškama je vraćanje *koda greške* iz pozvane funkcije i prepuštanje odluke o daljim koracima funkciji pozivniku

Upotreba izuzetaka

- Poseban način rukovanja greškama koje se ne očekuju u normalnom izvršavanju programa je pomoću *izuzetaka* (*exceptions*), npr.

```
double deljenje(float brojilac, float imenilac) {  
    if (imenilac == 0)  
        throw "GREŠKA: deljenje s nulom nije definisano!\n";  
    else  
        return static_cast<double> brojilac / imenilac;  
}
```

- uzrok ovakvih greški najčešće nije u kodu programa, već u kodu koji programeru nije dostupan (biblioteke), pa se uobičajeni načini otklanjanja grešaka ne mogu koristiti
- prednost upotrebe izuzetaka za prijavu grešaka je potpuno odvajanje koda koji otklanja grešku od onog koji je grešku prouzrokovao

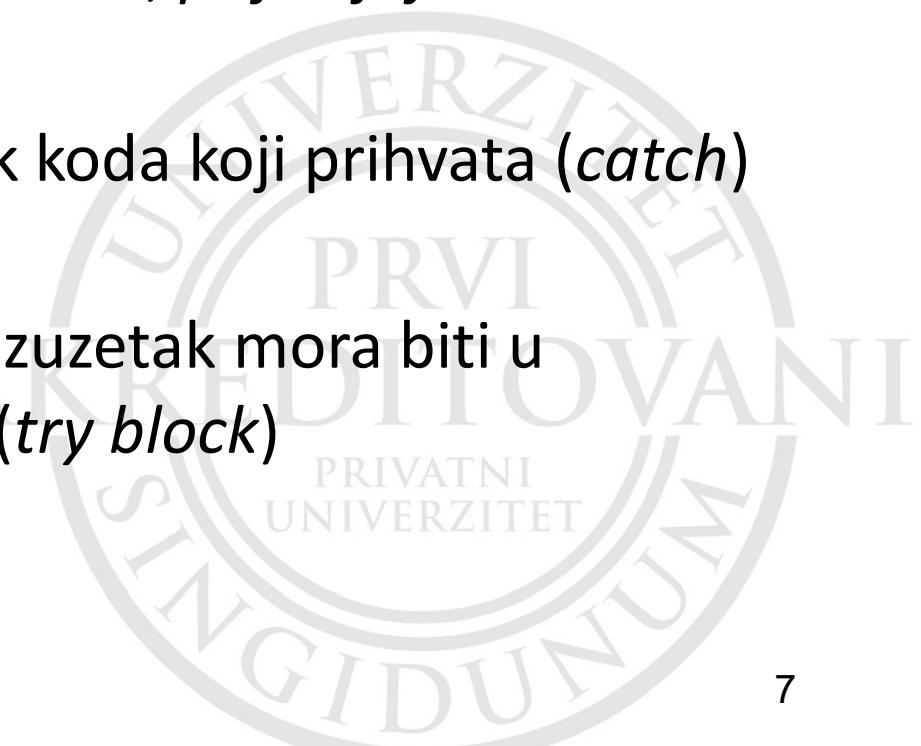
2. Rukovanje izuzecima

- 1. Prijava izuzetka**
- 2. Rukovanje izuzecima**
- 3. Kod koji prijavljuje izuzetak**
- 4. Ugnježdeni blokovi za rukovanje izuzecima**



2.1 Prijava izuzetka

- Izuzetak u jeziku C++ je privremeni objekt *bilo kog tipa* koji se koristi za prijavu greške
 - izuzetak može biti osnovnog tipa, npr. `int` ili `const char*`, ali se češće definiše kao objekt tipa neke *klase*
- Kad se u programu otkrije neki problem, *prijava*juje se ili *baca* izuzetak (*throwing exception*)
- Kontrola se prenosi u poseban blok koda koji prihvata (*catch*) i obrađuje izuzetak
- Programski kod koji može prijaviti izuzetak mora biti u posebnoj vrsti programskog bloka (*try block*)



Sintaksa bloka za prijavu izuzetka

- Blok naredbi koji može da prijavi izuzetak i naredbe za prihvatanje i obradu izuzetka strukturiraju se kao:

```
try {  
    // Kod programa koji može da prijavi izuzetak  
}  
catch (parametar koji navodi izuzetak tipa 1) {  
    // Kod za rukovanje izuzetkom  
}  
...  
catch (parametar koji navodi izuzetak tipa n) {  
    // Kod za rukovanje izuzetkom  
}
```

- programski kod u blokovima za rukovanje izuzecima (*catch*) izvršava se samo u slučaju pojave izuzetka *odgovarajućeg tipa*

Prijava izuzetaka

- Jednostavan primer prijave izuzetaka pomoću osnovnog tipa podataka:

```
try {  
    // Kod koji može da prijavi izuzetak je u try bloku  
    if (test > 5)  
        throw "test je veće od 5"; // prijava izuzetka tipa const char*  
    // Ostatak programa koji se izvršava ako nije prijavljen izuzetak  
}  
catch(const char* poruka) {  
    // Kod za rukovanje izuzetkom, koji se izvršava  
    // ako je prijavljen izuzetak tipa 'char*' ili 'const char'*  
    cout << poruka << endl;  
}
```

- ako je vrednost promenljive `test > 5`, prekida se normalno izvršavanje programa i obrađuje izuzetak (poruka o grešci "test je veće od 5")

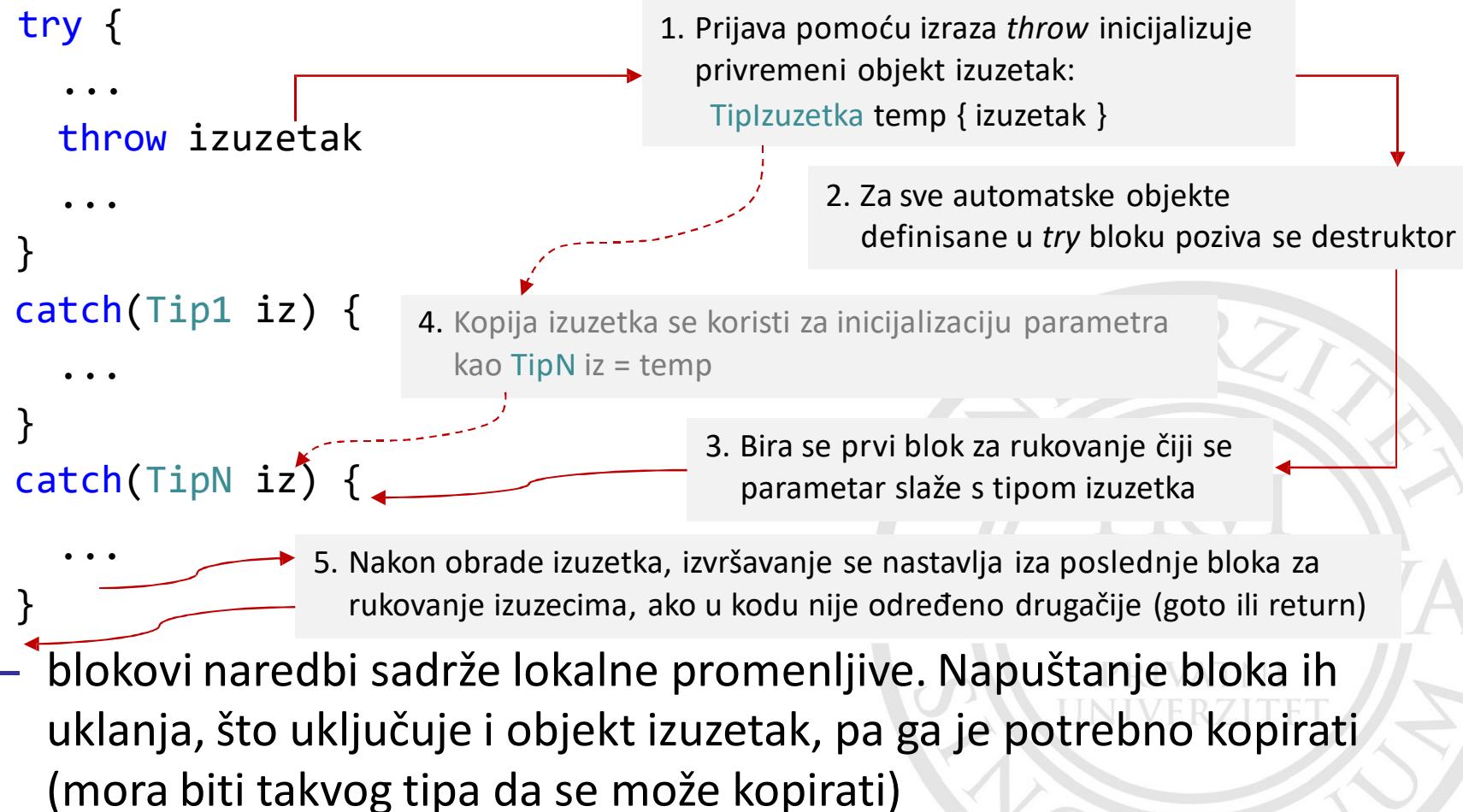
Primer: Upotreba osnovnih tipova podataka za prijavu izuzetaka

```
#include <iostream>
using namespace std;
int main() {
    for (size_t i=0; i < 7 ; ++i) {
        try {
            if (i < 3) throw i;
            cout << "Nije prijavljen izuzetak za i= " << i << endl;
            if (i > 5) throw "Jos jedan izuzetak!";
            cout << "Kraj try bloka." << endl;
        }
        catch (size_t i) { // hvatanje izuzetka tipa size_t
            cout << "Uhvacen izuzetak za i= " << i << std::endl;
        }
        catch (const char* poruka){ // hvatanje izuzetka tipa char*
            cout << " \\" << poruka << "\"" uhvacen" << endl;
        }
        cout << "Kraj petlje nakon catch blokova, i= " << i << endl;
    }
    return 0;
}
```

```
Uhvacen izuzetak za i= 0
Kraj petlje nakon catch blokova, i= 0
Uhvacen izuzetak za i= 1
Kraj petlje nakon catch blokova, i= 1
Uhvacen izuzetak za i= 2
Kraj petlje nakon catch blokova, i= 2
Nije prijavljen izuzetak za i= 3
Kraj try bloka.
Kraj petlje nakon catch blokova, i= 3
Nije prijavljen izuzetak za i= 4
Kraj try bloka.
Kraj petlje nakon catch blokova, i= 4
Nije prijavljen izuzetak za i= 5
Kraj try bloka.
Kraj petlje nakon catch blokova, i= 5
Nije prijavljen izuzetak za i= 6
"Jos jedan izuzetak!" uhvacen
Kraj petlje nakon catch blokova, i= 6
```

2.2 Rukovanje izuzecima

- Proces rukovanja izuzecima odvija se kao na prikazu:



Neobrađeni izuzeci

- Ako nije pronađen kod za rukovanje (*catch* blok) za neki prijavljeni izuzetak pokreće se funkcija `std::terminate()` definisana u zaglavlju `<exception>`
 - ova funkcija poziva *podrazumevanu* funkciju za rukovanje izuzetkom `std::abort()` definisanu u zaglavlju `<cstdlib>`, koja odmah terminira program i ne poziva destruktore automatskih i statičkih objekata
 - bolji način je promena podrazumevane funkcije u drugu *standardnu* funkciju `exit()` pomoću funkcije `set_terminate()`, npr.

```
void myHandler() {  
    exit(1); // oslobađa memoriju pre terminiranja  
}  
...  
terminate_handler pOldHandler = set_terminate(myHandler);
```

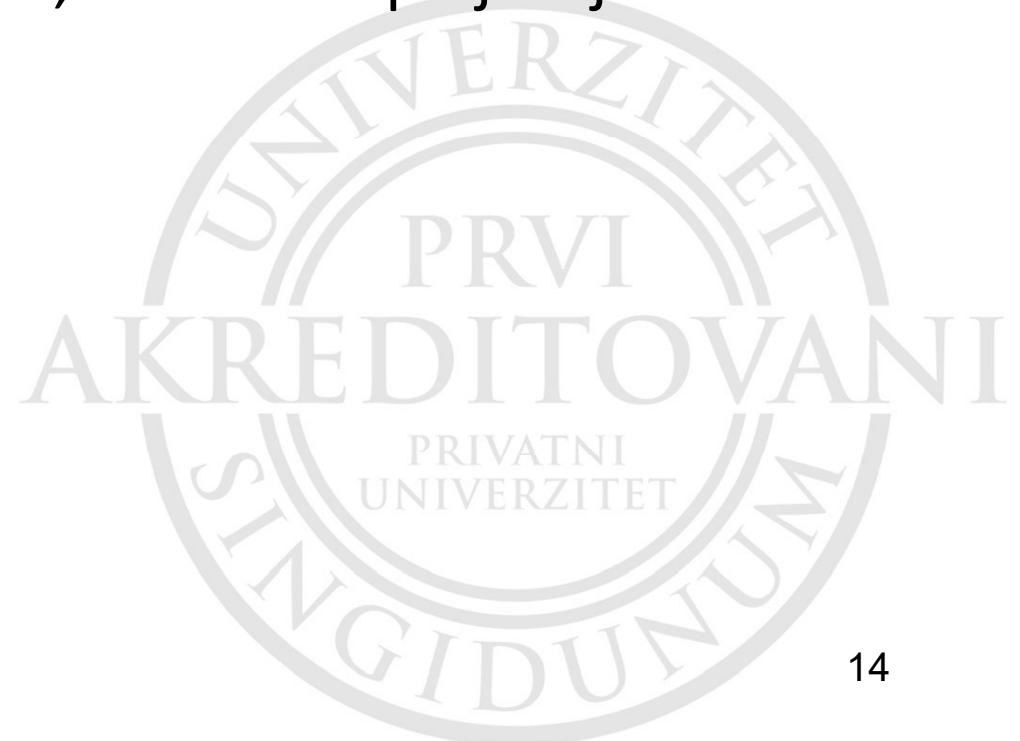
2.3 Kod koji prijavljuje izuzetak

- Kod koji prijavljuje izuzetak može se samo logički nalaziti u **try** bloku, a fizički može biti bilo gde u programu
- Isti programski kod (funkcija) može biti pozvan iz različitih **try** blokova, tako da se izuzetak koji se prijavljuje može, u različito vreme, obraditi u sasvim različitim **catch** blokovima



2.4 Ugnježdeni blokovi za rukovanje izuzecima

- Blokovi naredbi za prijavu izuzetaka mogu se gnezdit
- Svaki **try** blok za prijavu izuzetaka ima sopstvene blokove za njihovu obradu u kojima se prvo traži odgovarajući **catch** blok
- Ukoliko se odgovarajući **catch** blok za obradu izuzetka ne pronađe u unutrašnjem **try** bloku, traži se u spoljašnjem bloku

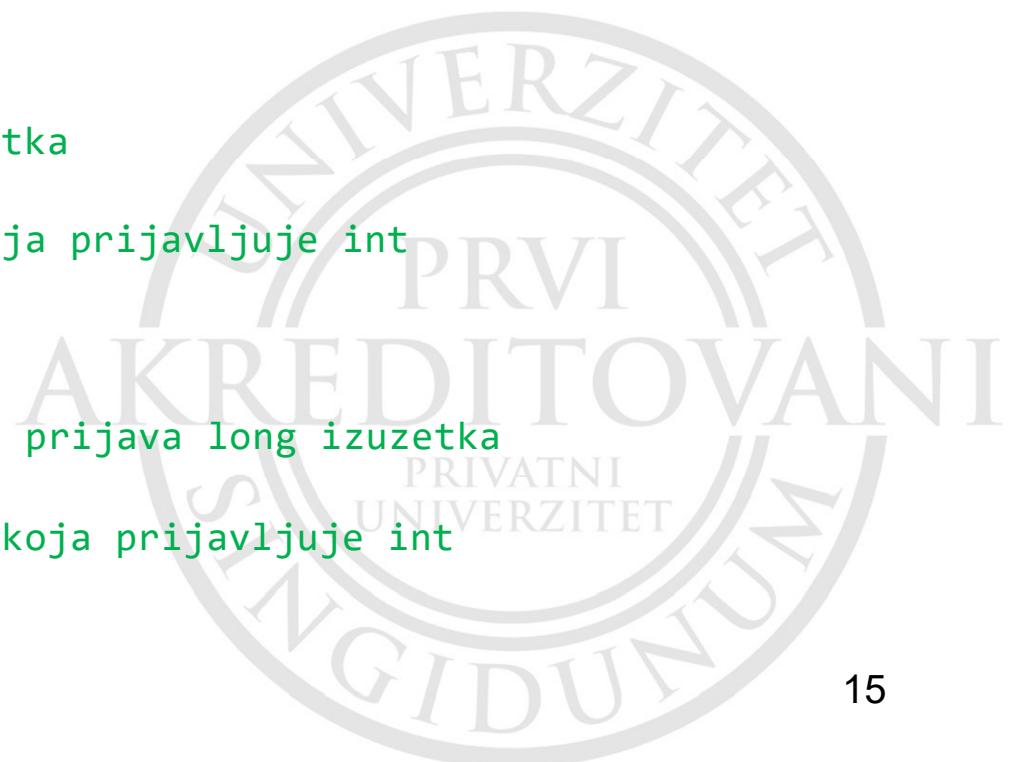


Primer: Ugnježdeni blokovi za prijavu izuzetaka (1/2)

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

void throwIt(int i) {
    throw i; // prijavljuje vrednost parametra
}

int main() {
    for (int i=0; i <= 5; ++i) {
        try {
            cout << "Spoljni try:\n";
            if (i == 0)
                throw i; // prijava int izuzetka
            if (i == 1)
                throwIt(i); // poziv funkcije koja prijavljuje int
            try { // ugnježdeni try blok
                cout << " Unutrasnji try:\n";
                if (i == 2)
                    throw static_cast<long>(i); // prijava long izuzetka
                if (i == 3)
                    throwIt(i); // poziv funkcije koja prijavljuje int
            } // kraj ugnježdenog try bloka
        }
    }
}
```



Primer: Ugnježdeni blokovi za prijavu izuzetaka (2/2)

```
catch (int n) {
    cout << " Catch int za unutrasnji try. "
        << "Izuzetak " << n << endl;
}
cout << "Spoljni try:\n";
if (i == 4)
    throw i; // prijava int
throwIt(i); // poziv funkcije koja prijavljuje int
}
catch (int n) {
    cout << "Catch int za spoljni try. "
        << "Izuzetak " << n << endl;
}
catch (long n) {
    cout << "Catch long za spoljni try. "
        << "Izuzetak " << n << endl;
}
}
return 0;
```

```
Spoljni try:
Catch int za spoljni try. Izuzetak 0
Spoljni try:
Catch int za spoljni try. Izuzetak 1
Spoljni try:
Unutrasnji try:
Catch long za spoljni try. Izuzetak 2
Spoljni try:
Unutrasnji try:
Catch int za unutrasnji try. Izuzetak 3
Spoljni try:
Catch int za spoljni try. Izuzetak 3
Spoljni try:
Unutrasnji try:
Spoljni try:
Catch int za spoljni try. Izuzetak 4
Spoljni try:
Unutrasnji try:
Spoljni try:
Catch int za spoljni try. Izuzetak 5
```

3. Objekti klase kao izuzeci

1. Prihvatanje izuzetka
2. Prihvatanje izuzetaka izvedene klase rukovaocem osnovne klase
3. Ponovljeno prijavljivanje izuzetaka (rethrowing)
4. Prihvatanje svih izuzetaka



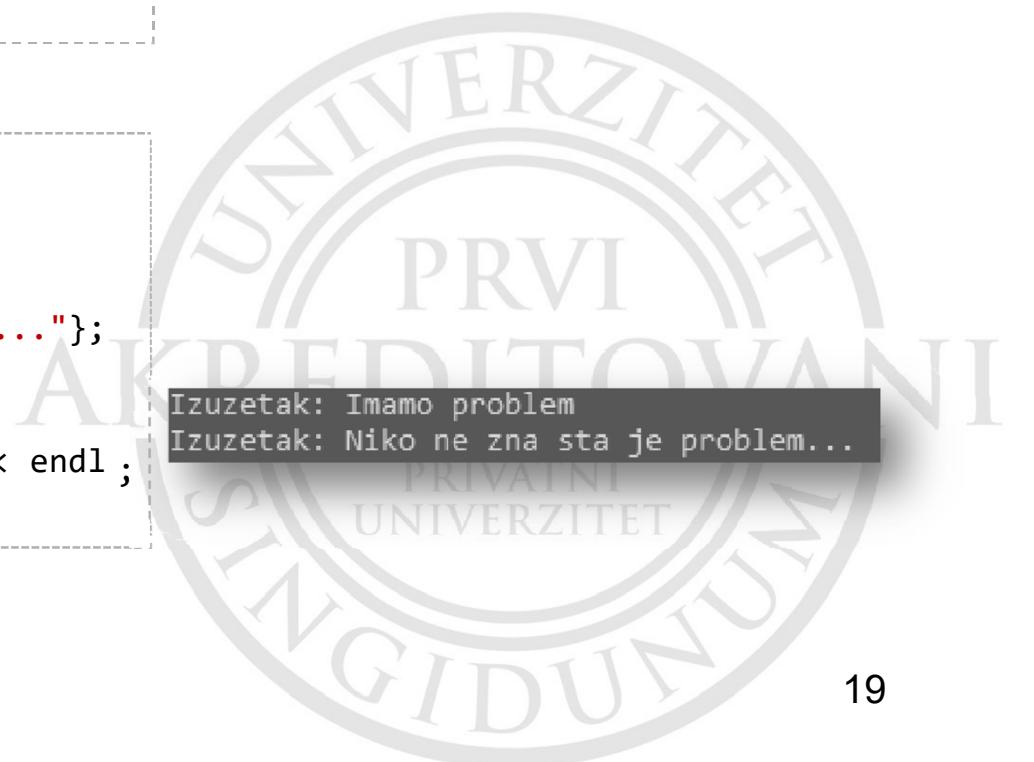
3.1 Prihvatanje izuzetka

- Izuzetak je objekt bilo koje klase, čija je osnovna namena *prenos informacija* programu za obradu izuzetka
- Može se definisati posebna *korisnička klasa izuzetaka*, koja može da sadrži informacije o problemu, kod greške i neke dodatne informacije, npr.

```
class Problem {  
    private:  
        string poruka;  
    public:  
        Problem(string s="Imamo problem") : poruka {s} {}  
        string prikaziProblem() const { return poruka; }  
};
```

Primer: Upotreba korisničke klase izuzetaka

```
#include <iostream>
#include <string>
using namespace std;
class Problem {
private:
    string poruka;
public:
    Problem(string s="Imamo problem") : poruka {s} {}
    string prikaziProblem() const { return poruka; }
};
int main() {
    for(int i=0; i < 2 ; ++i) {
        try {
            if (i == 0)
                throw Problem {};
            else
                throw Problem {"Niko ne zna sta je problem..."};
        }
        catch (const Problem& t) {
            cout << "Izuzetak: " << t.prikaziProblem() << endl ;
        }
    }
    return 0;
}
```



```
Izuzetak: Imamo problem
Izuzetak: Niko ne zna sta je problem...
```

Proces prihvatanja izuzetka

- Prilikom prihvatanja izuzetka neophodno je *poklapanje* tipa izuzetka i parametra u **catch** bloku
- Prilikom poklapanja su moguće situacije:
 - tip parametra je *isti* kao tip izuzetka (ako se zanemari oznaka **const**)
 - tip parametra je *osnovna klasa* tipa klase izuzetaka ili *referenca* na osnovna klasu tipa klase izuzetaka, direktno ili indirektno (ako se zanemari oznaka **const**)
 - izuzetak i parametar su *pokazivači*, a tip izuzetka se može *implicitno konvertovati* u tip parametra (ako se zanemari oznaka **const**)
- Ako postoji hijerarhija klasa izuzetaka, potrebno je prvo navesti blok za najnižu izvedenu klasu, a na kraju za osnovnu klasu, inače će se uvek poklopiti osnovna, a ne izvedena klasa

3.2 Prihvatanje izuzetaka izvedene klase rukovaocem osnovne klase

- Izuzeci izvedene klase izuzetaka implicitno se konvertuju u tip *osnovne klase* radi poklapanja parametra **catch** bloka, tako da se više prijavljenih izuzetaka može uhvatiti jednim rukovaocem
- Parametar se inicijalizuje pomoću konstruktora kopije *osnovne klase*, čime se gube svojstva *izvedene klase*
- Pojava gubitka informacija o izvedenim klasama naziva se *object slicing* i predstavlja opšti izvor grešaka prilikom prenošenja objekata po vrednosti
- Zbog toga je u **catch** blokovima neophodno uvek koristiti parametre tipa *reference*

3.3 Ponovljeno prijavljivanje izuzetaka (*rethrowing*)

- Kada rukovalac unutrašnjeg bloka prihvati izuzetak, može ga ponovo prijaviti radi omogućavanja obrade izuzetka u spoljašnjem bloku pomoću posebne forme naredbe, koja sadrži samo reč
throw;
- Tada se ponovo prijavljuje tekući izuzetak, odnosno *postojeći objekt* izuzetka, bez ponovnog kopiranja, što je važno ako je u pitanju izvedena klasa izuzetaka, koja inicijalizuje parametar osnovne klase (referencu)
- Ponovljenu prijavu izuzetka ne koriste ostali rukovaoci unutrašnjeg bloka

3.4 Prihvatanje svih izuzetaka

- Specifikacija parametara pomoću *tri tačke* u *catch* bloku (...) označava da blok može da rukuje bilo kojim izuzetkom:

```
try {  
    // Programska koda koji može da prijavi izuzetak  
}  
catch (...) {  
    // Programska koda za rukovanje bilo kojim izuzetkom  
}
```

- Rukovalac koji prihvata sve izuzetke treba da bude poslednji u bloku

4. Funkcije koje prijavljuju izuzetke

1. Funkcije za prijavu izuzetaka
2. Funkcije koje ne prijavljuju izuzetke
3. Prihvatanje izuzetaka u konstruktoru
4. Izuzeci i destruktori



4.1 Funkcije za prijavu izuzetaka

- Svaka funkcija može da prijavi izuzetak, uključujući konstruktore klase. Izuzetak koji funkcija prijavi može se prihvati u funkciji *pozivniku*, ali je tada potrebno da se u samoj funkciji *ne* prihvati ili se ponovo prijavi (*rethrow*)
 - izuzetak je potrebno negde prihvati da se spreči terminiranje programa zbog neobrađenih izuzetaka
 - poziv funkcije koja prijavljuje izuzetke neophodno je zatvoriti u **try** blok koji prihvata izuzetke
 - telo funkcije može da sadrži **try** i **catch** blokove. Svaki izuzetak koji nije prihvaćen prosleđuje se do tačke gde je funkcija pozvana
- Ponekad je pogodno da celo telo funkcije bude **try** blok sa skupom rukovalaca (*function try block*)

Ilustracija: Funkcionalni blokovi za prihvatanje izuzetaka (*function try blocks*)

- Ključna reč **try** može se postaviti *ispred*, a **catch** blokovi *iza* tela funkcije, tako da je kompletno telo funkcije *try-catch* blok
 - ako se izvršavanje programa ne prekida, **catch** blok treba da završi naredbom **return**
 - za tip **void** kraj **catch** bloka je ekvivalent izvršavanja naredbe **return**

```
void funkcijaUradi(int argument)
try {
    // Telo funkcije (kod)
}
catch (VelikiProblem& iz){
    // Obrada izuzetka VelikiProblem
}
catch (VeciProblem& iz){
    // Obrada izuzetaka VeciProblem
}
catch(Problem& iz) {
    // Obrada izuzetaka tipa Problem
}
```

4.2 Funkcije koje ne prijavljuju izuzetke

- Pomoću specifikacije `noexcept` u zagлавju funkcije može se označiti da funkcija prihvata, a ne ponavlja prijave izuzetaka, npr.

```
void funkcijaUradi(int argument) noexcept
try {
    // Programska logika funkcije
}
catch ( ... ) {
    // Obrada svih izuzetaka (ne prijavljuju se ponovo)
}
```

- Funkcija obrađuje sve izuzetke koje može da uhvati. Ako se *prijava* izuzetak koji nije uhvaćen u funkciji, neće se proslediti funkciji pozivniku, već će se odmah pozvati `std::terminate()`

4.3 Prihvatanje izuzetaka u konstruktoru

- Konstruktorske funkcije mogu da prijave izuzetak i bez eksplisitne naredbe **throw** u telu konstruktora
 - npr. zbog upotrebe funkcija iz standardne biblioteke ili nekih funkcija klase **string**, koje *mogu da prijave* izuzetke, ako konstruktor koristi **new**
- Ako je izuzetak prijavljen i prihvaćen u konstruktoru, **catch** blok treba da ponovi prijavu izuzetka (*rethrow*), da se pozivnik obavesti o tome da objekt nije bio izgrađen
- Ako izvršavanje programa dođe do kraja **catch** bloka bez ponavljanja prijave izuzetka, ponoviće se prijava originalnog izuzetka

Izuzetak u listi inicijalizacije

- Ako postoji mogućnost da konstruktor prilikom inicijalizacije prijavi izuzetak, *lista inicijalizacija* se postavlja u **try** blok, koji se navodi odmah nakon liste parametara, npr.

```
Primer::Primer(int brojac) try : OsnovnaKlase(brojac) {  
    // Konstruktor osnovne klase može da prijavi izuzetak  
}  
catch(...) // blok prihvata sve izuzetke {  
    // Obrada izuzetka  
    throw; // nije neophodno  
}
```

- Klasa je izvedena iz klase osnovne klase, pa konstruktor klase **Primer** poziva konstruktor osnovne klase u *listi inicijalizacije*

4.4 Izuzeci i destruktori

- Destruktori ne treba da prijavljuju izuzetke, po definiciji su **noexcept**, tako da pojava izuzetka terminira program
- Za automatski kreirane objekte može se, prilikom pojave izuzetka, dogoditi da se destruktur klase pokrene *pre* obrade izuzetka u **catch** bloku
- Destruktur može sam da ustanovi situaciju da je pozvan usled pojave izuzetka pre izvršenja **catch** bloka funkcijom **std::uncaught_exceptions()**
- Ako je neophodno sprečavanje pojave izuzetaka izvan granica destruktora, kod destruktora se može zatvoriti u **try** blok čiji **catch** blok prihvata sve izuzetke

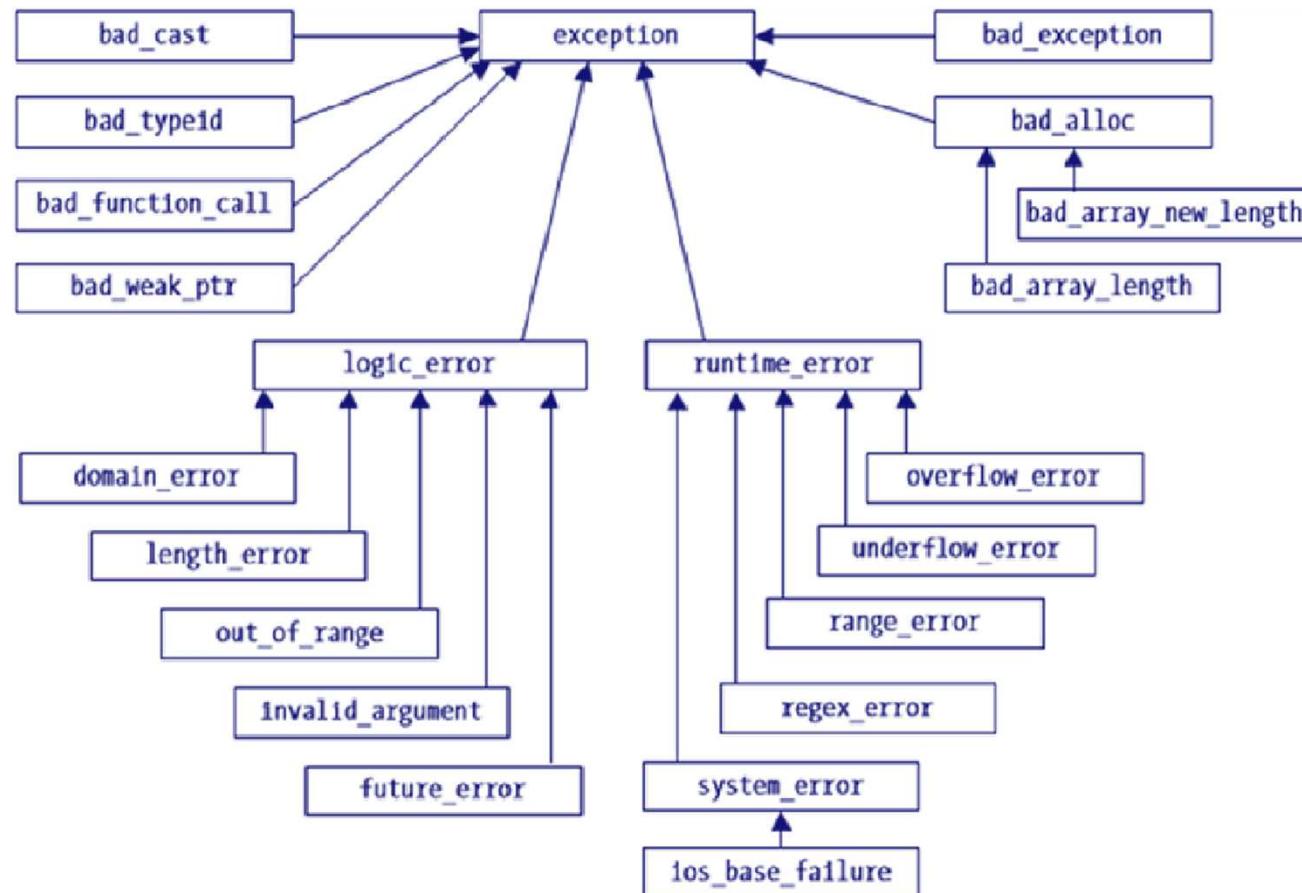
5. Standardna biblioteka izuzetaka

- 1.** Standardni tipovi klasa izuzetaka
- 2.** Upotreba standardnih izuzetaka



5.1 Standardni tipovi klasa izuzetaka

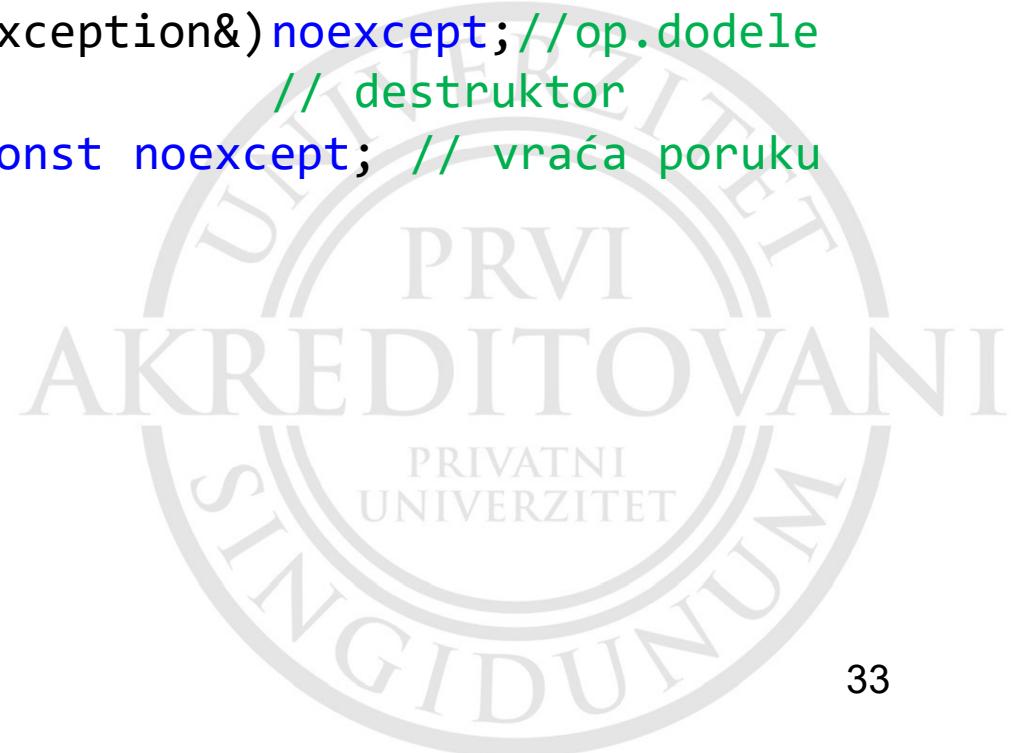
- Standardna biblioteka sadrži definicije nekih klasa izuzetaka, a izvedene klase definisane su u zaglavlju `<stdexcept>`



Definicija osnovne klase izuzetaka

- Klasa izuzetaka `exception` može se koristiti za izvođenje sopstvenih klasa izuzetaka, koje nasleđuju sve članove:

```
class exception {  
public:  
    exception() noexcept;                      // podraz. konstruktor  
    exception(const exception&) noexcept;        // konstruktor kopije  
    exception& operator=(const exception&) noexcept; // op.dodele  
    virtual ~exception();                         // destruktur  
    virtual const char* what() const noexcept; // vraća poruku  
};
```



5.2 Upotreba standardnih izuzetaka

- Standardne klase izuzetaka mogu se koristiti na dva načina:
 - za prijavu standardnih tipova izuzetaka
 - za kreiranje sopstvenih klasa izuzetaka, koje se izvode iz standardnih
- Npr. u konstruktoru klase **Kutija** može se prijaviti izuzetak *range_error* ako se zadaju nedozvoljene dimenzije kutije:

```
Kutija::Kutija(double d, double s, double v) : duzina {d},  
sirina {s}, visina {v} {  
    if (d <= 0.0 || s <= 0.0 || v <= 0.0)  
        throw range_error("Dimenziye  
        <= 0");  
}
```

- U program treba uključiti zaglavljje **<stdexcept>** u kome je definisana klasa *range_error*

Kreiranje sopstvene klase izuzetaka

- Sopstvena klasa izuzetaka može se izvesti iz standardnih klasa izuzetaka, tako da se izuzeci sopstvene klase mogu hvatati u okviru istog **catch** bloka kao i standardni izuzeci
- Npr. za klasu **Kutija** može se iz standardne klase **range_error** izvesti klasa izuzetaka **Dim_error**, s detaljnijim opisom greške

```
#include <stdexcept>
#include <string>
using namespace std;

class Dim_error : public range_error {
public:
    using range_error::range_error;// nasleđ. osn. konstr.
    Dim_error(string s, int d) range_error{s+to_string(d)};
}
```

detaljniji opis greške: tekst i pogrešna vrednost

6. Primeri

- Izuzeci kod prenosa argumenata



Izuzeci kod prenosa argumenata [4]

```
// Izuzetak za nedozvoljenu
// vrednost argumenta funkcije

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

// Funkcija računa površinu kruga
double getPovrsina(double r) {
    if (r < 0)
        throw invalid_argument(
            "Poluprecnik ne može biti < 0!");

    return r * r * 3.14159;
}
```

```
int main() {
    // Unos poluprečnika
    cout << "Unesite poluprecnik: ";
    double poluprecnik;
    cin >> poluprecnik;

    try {
        double P = getPovrsina(poluprecnik);
        cout << "Povrsina= " << P << endl;
    }
    catch (exception& izuzetak) {
        cout << izuzetak.what() << endl;
    }

    cout << "Kraj." << endl;
    return 0;
}
```

Unesite poluprecnik: -5
Poluprecnik ne može biti < 0!
Kraj.

Literatura

1. Branović I., *Osnove objektno orijentisanog programiranja: C++*, Univerzitet Singidunum, 2013
2. Stroustrup B., *The C++ Programming Language*, 4th Ed, Addison Wesley, 2013
3. Horton I., Van Weert P., *Beginning C++ 20*, 6th Edition, Apress, 2020
4. Liang D., *Introduction to Programming With C++*, 3rd Ed, Pearson Education, 2014
5. Horton I., *Beginning C++*, Apress, 2014
6. Horton I., *Using the C++ Standard Template Libraries*, Apress, 2015
7. Horstmann C., *Big C++*, 2nd Ed, John Wiley&Sons, 2009
8. Veb izvori
 - <http://www.cplusplus.com/doc/tutorial/>
 - <http://www.learncpp.com/>
 - <http://www.stroustrup.com/>
9. Knjige i priručnici za *Visual Studio 2010/2012/2013/2015/2017/2019*