Tema 03

Funkcije, prenos parametara i dinamička alokacija memorije

Prof. dr Miodrag Živković

Tehnički fakultet

OBJEKTNO ORIJENTISANO PROGRAMIRANJE 2

Sadržaj

1. Uvod

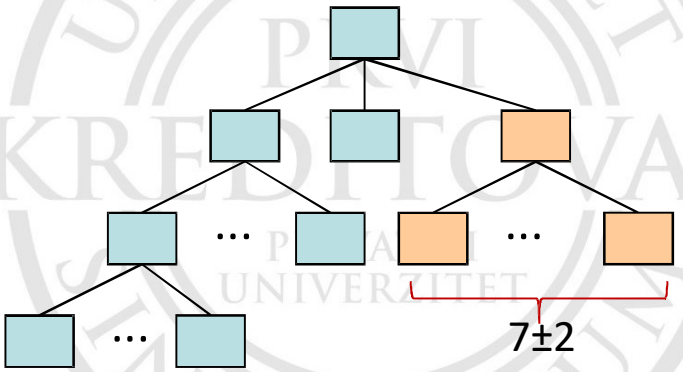
- Ponavljanje: pojam funkcije
- Ponavljanje: dekompozicija programskog koda

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 šta 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
```

parametri funkcije

aktuelni argumenti

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 uklanjaju
- 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;
  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;
  cout << "Vrednost promenljive u funkciji = " << promenljiva << endl;
}
```

promenljiva *je vidljiva u funkciji main()*

promenljiva *je vidljiva u funkciji mojaFunkcija()*

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 inicijalizuju 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

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 << ".";
}
```

promenljiva je vidljiva celom programu

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;
}
```

Rezultat:

```
100
```

parametar funkcije definisan kao pokazivač;
vrednost se dodeljuje promenljivoj na koju pokazuje

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*.

  Eksplicitnim korišćenjem pokazivača moguće je ostvariti prenos argumenata *po adresi* (referenci), tako da se prenose adrese aktuelnih argumenata umesto njihovih vrednosti
- Eksplicitna 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 poprimi vrednost NULL, a referenca uvek pokazuje na određeni objekt
  - pristup objektu preko pokazivača vrši se posredno pomoću operatora *, a pristup preko reference je direktan
  - 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 referenca 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
```

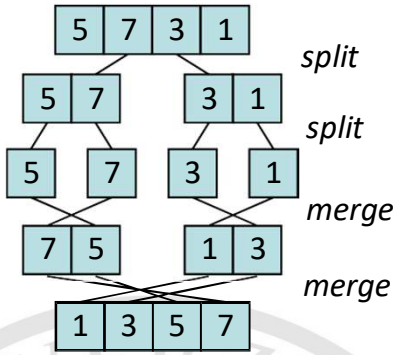
8. Primeri programa

1. Sortiranje polja: Merge sort
2. Najkraći put u grafu: Dijkstrin algoritam

8.1 Sortiranje polja: Merge sort

John von Neumann, 1945

- Rekurzivni 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)

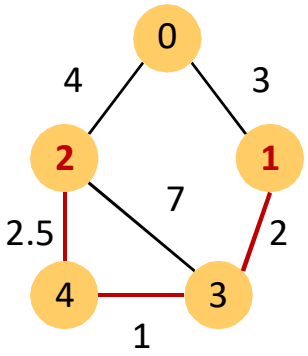
```
#include <iostream>
using namespace std;

// 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++;
  }
}
```

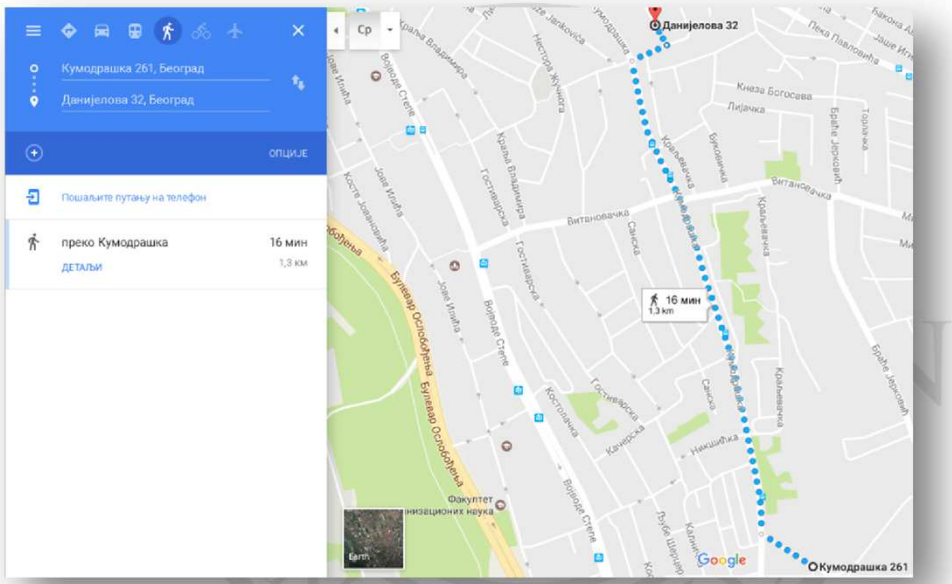
Program (2/2)

```
// 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)

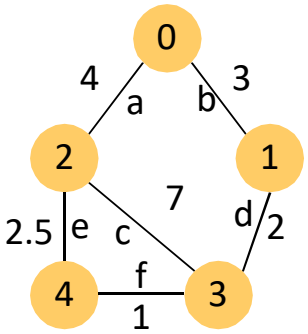
- Primer upotrebe matrica u rešavanju grafovskih problema
  - graf $G(v,e)$ je skup čvorova v (*vertex, node*) povezanih lukovima e (*edge*)
  - osim oznaka, *lukovi* grafa mogu imati i različita kvantitativna obeležja, koja predstavljaju npr. cenu, dužinu i sl.
  - pretraživanjem je moguće pronaći *najkraći 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 << "Cvor 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 izme-u 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 |
       | | 7   |      | -  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