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 upotrebe 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

Napomena:

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 porediti 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
    // Zapremina 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 poredi 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 inicijalizacija 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;
}
```

```
Najmanja kutija ima dimenzije:1x2x1
```

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 predvidi 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();                         // destruktor
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; //ne 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

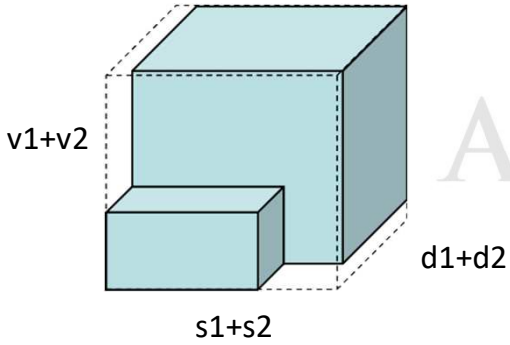
```
if (this == &poruka) return *this //samo vraća levi operand
```

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

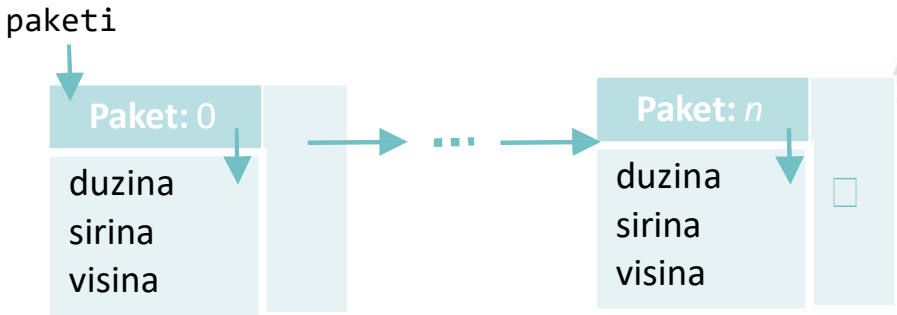


4.2 Preklapanje operatora poređenja

- 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 zapremine (*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. Virtelni 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 prevođenja 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 prevođenja

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 zapremine objekta
  void prikaziZapreminu() const {
    cout << "Kutija ima korisnu zapreminu " << zapremina() << endl;
  }
  // Funkcija 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; // direktan 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 nephodne 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 prevođenja

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 oslobodi(Osnovna *p) {
  delete p; // pokreće se virtuelni destruktor
}
int main() {
  Osnovna *posn = new Osnovna;   Izvedena *pizv = new Izvedena;
  oslobodi(posn); oslobodi(pizv);
  return 0;
}
```

```
Virtuelni destruktor osnovne klase
Virtuelni destruktor izvedene klase
Virtuelni destruktor osnovne klase
```

7. Konverzije tipova

- Konverzija između pokazivača i objekata klasa
- Dinamička konverzija
- Konverzija referenci
- Ustanovljavanje polimofnog tipa

Konverzija između pokazivača i objekata klasa

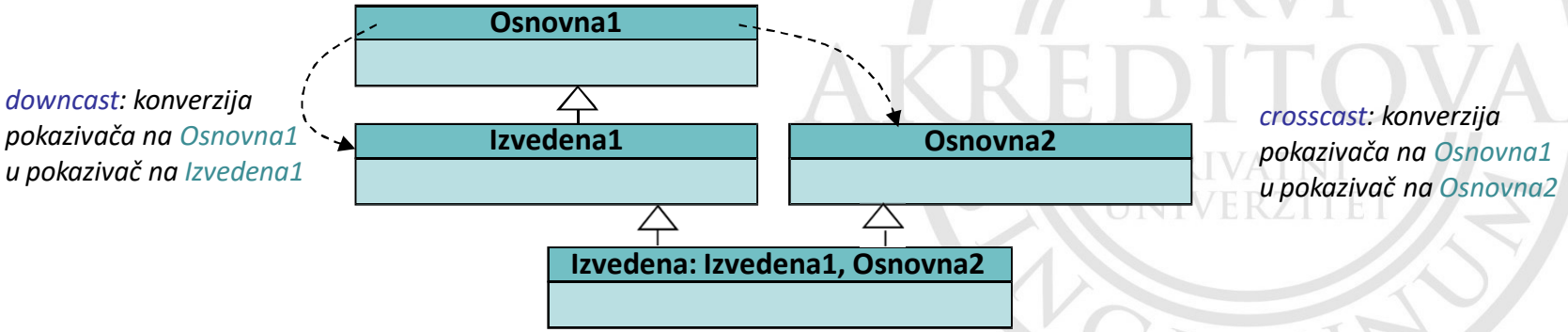
- Moguće je izvršiti *implicitnu konverziju* pokazivača na izvedenu klasu u pokazivač na osnovnu klasu, kako za direktno, tako i za indirektno nasleđene klase
- Np. pokazivač na klasu KartonskaKutija može se implicitno konvertovati:

```
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 virtuelnu 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 zaglavlju 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