

Klase (class)

Objektno programiranje - 6. vježbe

dr. sc. Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

9. travnja 2024. godine



Klase - definiranje vlastitih tipova podataka

- ▶ želimo def. tipove koji se ponašaju prirodno poput ugrađenih
- ▶ važno: ime, koje operacije podržava i gdje je def.

Primjer.

```
#include <iostream> //zaglavlje iz stand. bibl.  
#include "Polinom.h" //nije iz stand. bibl.  
  
...  
Polinom p, q;  
cin >> p >> q;  
cout << p + q;  
...
```



Primjer - izgradnja klase za račun klijenta u banci

Klijenta banke želimo identificirati id-jem (npr. "642986"), pratiti koliko ima na tekućem računu i koliko mu je dopušteno prekoračenje po tekućem računu.

```
struct Racun {  
    string id;  
    double saldo;  
    double prekoracenje;  
};
```

Primjer.

```
Racun a;  
a.prekoracenje = -1327.23;  
cout << a.prekoracenje << " EUR" << endl;
```



in-class inicijalizacija

- ▶ ako ne želimo *defaultne* vrijednosti pri stvaranju objekta (s {} ili =)

```
struct Racun {  
    string id;  
    double saldo = -3.98;  
    double prekoracenje{-1327.23};  
};
```

Primjer.

```
Racun a, b;  
a = b;  
cout << b.id << endl  
    << b.saldo << endl  
    << b.prekoracenje << endl;
```



Pisanje vlastitih zaglavlja

- ▶ datoteka **Racun.h** (za korištenje klase u više datoteka)
- ▶ uočite **zaštitu protiv višestrukog uključivanja**
- ▶ uočite da nemamo `using namespace std;`

```
#ifndef RACUN_H
#define RACUN_H
#include <string>
struct Racun {
    std::string id;
    double saldo = -3.98;
    double prekoracenje{-1327.23};
};
#endif
```

- ▶ upotreba:

```
#include "Racun.h"
...
Racun a, b;
```



Pisanje sučelja (*interface*): Funkcije članice

- ▶ deklarirane unutar, a definirane unutar (tada su *inline*) ili izvan tijela klase

```
struct Racun {
    std::string poziv_na_broj() {return "0000"+id;}
    std::string id;
    double saldo = -3.98;
    double prekoracenje{-1327.23};
};
```

ili:

```
struct Racun {
    std::string poziv_na_broj();
    ...
};
std::string Racun::poziv_na_broj() {
    return "0000" + id;
}
```



Implicitan parametar funkcije: `this`

- ▶ po *defaultu* konstantan pokazivač na nekonst. verziju tipa klase
- ▶ prethodna funkcija je zapravo (s implicitnim parametrom `Racun *const this`):

```
std::string Racun::poziv_na_broj() {  
    return "0000" + this->id;  
}
```

Primjer. Zašto se sljedeći kod ne kompajlira?

```
Racun a;  
a.id = "1234";  
const Racun b = a;  
cout << b.id << endl;    ✓  
cout << b.poziv_na_broj() << endl;    ✗
```



Popravak: `const` funkcija članica

```
struct Racun {  
    std::string poziv_na_broj() const;  
    ...  
};  
std::string Racun::poziv_na_broj() const {  
    return "0000" + id;  
}
```

- ▶ sad ne možemo mijenjati `*this`:

```
std::string Racun::poziv_na_broj() const {  
    id = "abc";    ✗  
    return "0000" + id;  
}
```



Funkcija dodaj (funkcija koja „vraća this”)

```
struct Racun {
    Racun& dodaj(const Racun&);
    ...
    double saldo = -3.98;
};
Racun& Racun::dodaj(const Racun& desni) {
    saldo += desni.saldo;
    return *this;
}
```

Primjer.

```
Racun a, b;
b.saldo = 100;
a.dodaj(b).dodaj(b);
cout << a.saldo << endl;
```



Funkcije koje su samo konceptualno dio klase

- obično deklarirane (ali ne i definirane) u istom zaglavlju s klasom

```
#ifndef RACUN_H
#define RACUN_H
```

Sučelje: Racun.h

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

```
struct Racun {
    std::string poziv_na_broj() const;
    Racun& dodaj(const Racun&);
    std::string id;
    double saldo = -3.98;
    double prekoracenje{-1327.23};
};
```

```
std::istream &unos(std::istream &, Racun &);
std::ostream &ispis(std::ostream &, const Racun &);
#endif
```



Funkcije za unos i ispis

Implementacija: Racun.cpp

```
#include <iostream>
#include <string>
#include "Racun.h"
using namespace std;

istream &unos(istream &is, Racun &r) {
    is >> r.id >> r.saldo;
    return is;
}

ostream &ispis(ostream &os, const Racun &r) {
    os << r.id << " " << r.saldo << " "
        << r.prekoracenje;
    return os;
}

string Racun::poziv_na_broj() const {
    return "0000" + id;
}
```



Implementacija (nastavak) i klijentski program

```
Racun& Racun::dodaj(const Racun& desni){
    saldo += desni.saldo;
    return *this;
}
```

Klijentski program: zad.cpp

```
#include<iostream>
#include<string>
#include "Racun.h"
using namespace std;

int main() {
    Racun a;
    unos(cin, a);
    ispis(cout, a) << endl;
    return 0;
}
```

Primjer. (unos i ispis):

```
ab12 100.34
ab12 100.34 -1327.23
```

```
g++ -Wall Racun.cpp zad.cpp -std=c++11 -o prog
```



Konstruktori

- ▶ isto ime kao klasa; ne mogu se deklarirati kao `const` (jer objekt svoju „konstantnost“ dobiva tek kad konstruktor završi)
- ▶ ako ne želimo *defaultni* konstruktor, možemo napisati svoj:

```
struct Racun {  
    Racun(const std::string &, double, double);  
    ...  
};
```

```
Racun::Racun(const string &s, double iznos,  
            double placa)  
{  
    id = s;  
    saldo = iznos;  
    prekoracenje = -1.5 * placa;  
}
```



Defaultni konstruktor

```
Racun a("A123", 100, 950.16);  
ispis(cout, a) << endl;  
Racun b;     X
```

- ▶ izgubili smo *defaultni* konstruktor - ako ga i dalje želimo:

```
struct Racun {  
    Racun() = default;  
    Racun(const std::string &, double, double);  
    ...  
};
```

- ▶ Napomena: Ako u klasi koristimo neku drugu klasu koja treba eksplicitni konstruktor, defaultni nam neće biti dobar!



Korištenje inicijalizacijske liste

```
Racun::Racun(const string &s, double iznos,  
             double placa)  
{  
    id = s;  
    saldo = iznos;  
    prekoracenje = -1.5 * placa;  
}
```

- ▶ umjesto prethodnog, možemo navesti članove strukture (ne nužno sve!) i njihove početne vrijednosti:

```
Racun::Racun(const string &s, double iznos,  
             double placa) : id(s), saldo(iznos),  
                             prekoracenje(-1.5 * placa) { };
```



Još dva konstruktora

```
struct Racun {  
    Racun(const std::string &);  
    Racun(std::istream &);  
    ...  
}
```

- ▶ jedan dobiva string za id, a drugi stream s kojeg čita podatke

```
Racun::Racun(const string &s) : id(s) { };
```

```
Racun::Racun(istream &is) {  
    unos(is, *this);  
}
```

```
Racun a("A123", 100, 950.16), b("B0"), c(cin);
```



Delegirajući konstruktor

- ▶ cijeli ili dio svog posla povjeravaju drugom konstruktoru
- ▶ mogli smo napisati ovako (drugi je delegirajući):

```
Racun::Racun(const string &s, double iznos,  
            double placa) : id(s), saldo(iznos),  
                           prekoracenje(-1.5 * placa) { };
```

```
Racun::Racun(const string &s) : Racun(s, 0, 1000) {  
};
```



Kontrola pristupa

- ▶ korisnik trenutno može mijenjati članove klase bez korištenja sučelja koje smo napisali

```
Racun a("E3A2");  
a.saldo = 5420.15;
```

- ▶ **public** članovi = sučelje (dostupno svim dijelovima programa)
- ▶ **private** članovi = enkapsulacija (skrivanje) implementacije (dostupno samo funkcijama članicama)

Koji je *defaultni* pristup:

- ▶ **struct** - javni
- ▶ **class** - privatni

Glavne prednosti enkapsulacije:

- ▶ korisnik nenamjerno pokvari stanje enkapsul. objekta
- ▶ promjena implementacije enkapsulirane klase ne mijenja kod kojim se korisnik koristi



Kontrola pristupa na našem primjeru

```
class Racun {
    public:
        Racun() = default;
        Racun(const std::string &, double, double);
        Racun(const std::string &);
        Racun(std::istream &);
        std::string poziv_na_broj() const;
        Racun& dodaj(const Racun&);
    private:
        std::string id;
        double saldo = -3.98;
        double prekoracenje{-1327.23};
};
```

```
Racun a("E3A2"), b;    ✓
a.dodaj(b);           ✓
b.saldo = 1500.50;    ✗
```



Napomena: Odlučivanje o kontroli pristupa

- ▶ ako želimo da funkcija `poziv_na_broj` nije namjenjena općoj upotrebi (tj. samo je dio implementacije) - i nju bi stavili kao privatnu

```
class Racun {
    public:
        ...
    private:
        std::string poziv_na_broj() const;
        ...
};
```



Problem. Kod se ne kompajlira - funkcije poput funkcije `unos` nisu članice klase, ali su dio sučelja:

```
istream &unos(istream &is, Racun &r) {
    is >> r.id >> r.saldo;
    return is;
}
```

Kako će klasa dopustiti drugoj funkciji pristup svojim `ne-public` članovima:

- ▶ dodamo deklaraciju te funkcije i ključnu riječ **friend** u klasu

Napomena: to ne znači da su to sad članovi naše klase ni da na njih utječe kontrola pristupa!



unos i ispis kao prijatelji naše klase

```
class Racun {
    friend std::istream &unos(std::istream &, Racun &);
    friend std::ostream &ispis(std::ostream &, const Racun &);
public:
    Racun() = default;
    Racun(const std::string &, double, double);
    Racun(const std::string &);
    Racun(std::istream &);
    std::string poziv_na_broj() const;
    Racun& dodaj(const Racun&);
private:
    std::string id;
    double saldo = -3.98;
    double prekoracenje{-1327.23};
};
```

```
std::istream &unos(std::istream &, Racun &);
std::ostream &ispis(std::ostream &, const Racun &);
```



Definiranje vlastitih lokalnih imena za tipove

- ▶ želimo da korisnik koristi to ime pa stavili `public`
- ▶ ne želimo da korisnik zna da koristimo `double` za iznose
- ⇒ skrivamo detalje implementacije od korisnika

```
class Racun {  
    ...  
    public:  
        typedef double valuta;  
        //ili: using valuta = double;  
        ...  
        Racun(const std::string &, valuta, valuta);  
        ...  
    private:  
        ...  
        valuta saldo = -3.98;  
        valuta prekoracenje{-1327.23};  
};
```



Definiranje vlastitih lokalnih imena za tipove (nastavak)

- ▶ promjena u `Racun.cpp`:

```
Racun::Racun(const string &s, valuta iznos,  
             valuta placa) : id(s), saldo(iznos),  
                           prekoracenje(-1.5 * placa) { };
```

- ▶ primjer upotrebe u klijentskom programu (`main` funkcija):

```
Racun::valuta placa;  
cin >> placa;  
Racun a("1234",0,placa);  
ispis(cout,a) << endl;
```



Umetnute (*inline*) funkcije

- ▶ manje funkcije je bolje imati kao *inline* - npr. `poziv_na_broj`
- ▶ po *defaultu* je tako ako ju definiramo unutar klase

```
class Racun {
    ...
    std::string poziv_na_broj() const {
        return "0000" + id;
    }
    ...
};
```

- ▶ umjesto toga, možemo (zbog preglednosti) eksplicitno deklarirati funkciju kao *inline*

```
class Racun {
    ...
    inline std::string poziv_na_broj() const;
    ...
};
```



Primjer, problem i rješenje

- ▶ u `main` funkciji:

```
Racun a("1234", 0, 1050.12);
cout << a.poziv_na_broj() << endl;
```

- ▶ problem pri kompajliranju:

In file included from zad.cpp:3:0:

*warning: inline function 'std::__cxx11::string
Racun::poziv_na_broj() const' used but never defined*

- ▶ isprobati: `g++ -c Racun.cpp, g++ -o prog Racun.cpp, g++ -c zad.cpp`
- ▶ rješenje: premjestimo iz `Racun.cpp` u `Racun.h` (usput morali dodati i `std::` prije `string`)

```
std::string Racun::poziv_na_broj() const {
    return "0000" + id;
}
```



Preopterećivanje funkcija članica

- ▶ isto ime (u istom doseg!) s različitim parametrima (broj/tipovi)
- ▶ ne želimo osmisliti (i zapamtiti) novo ime koje postoji samo zato da bi kompajler shvatio koju funkciju treba zvati

```
class Racun {  
    ...  
    Racun& dodaj(const Racun&);  
    Racun& dodaj(valuta);  
    ...  
};
```

```
Racun& Racun::dodaj(valuta br) {  
    saldo += br;  
    return *this;  
}
```



Primjer: koja funkcija se poziva?

- ▶ u funkciji `main`:

```
vector<Racun> v{Racun("1234", 0, 1050), Racun()};  
v[0].dodaj(v[1]).dodaj(100);  
ispis(cout, v[0]) << endl; //1234 96.02 -1575
```

- ▶ funkcije koje vraćaju referencu su lijeve vrijednosti (*lvalues* - ne vraćaju kopiju objekta)
- ⇒ gornjom konkatencijom izvršili na istom objektu više radnji



mutable članovi

- ▶ želimo pratiti u koliko je transakcija sudjelovao pojedini objekt:

```
class Racun {  
    ...  
    private:  
        size_t br_transakcija = 0;  
    ...  
};
```

```
Racun& Racun::dodaj(const Racun& desni) {  
    ++br_transakcija;  
    ++desni.br_transakcija; //Problem!  
    saldo += desni.saldo;  
    return *this;  
}  
Racun& Racun::dodaj(valuta br) {  
    ++br_transakcija;  
    saldo += br;  
    return *this;  
}
```



Rješenje: ključna riječ mutable

```
class Racun {  
    ...  
    private:  
        mutable size_t br_transakcija = 0;  
    ...  
};
```

```
ostream &ispis(ostream &os, const Racun &r) {  
    os << r.id << " " << r.saldo << " "  
    << r.prekoracenje << " ("  
    << r.br_transakcija << ")";  
    return os;  
}
```

```
Racun a("1234",0,1050.12),b; // zad.cpp  
a.dodaj(b);  
a.dodaj(100);  
ispis(cout,a) << endl; //1234 96.02 -1575.18 (2)  
ispis(cout,b) << endl; // -3.98 -1327.23 (1)
```



Funkcija za ispis stanja računa

- ▶ mogli bismo (**no nećemo!**) dodati sljedeću funkciju za ispis stanja na računu:

```
class Racun {  
    ...  
    public:  
        void stanje() const;  
    ...  
};
```

```
void Racun::stanje() const {  
    cout << "Stanje (" << id << ") = "  
        << saldo << " EUR." << endl;  
}
```

- ▶ sada bi za `Racun a("abc");` mogli napraviti

```
a.dodaj(100).stanje();    ✓
```

ali ne i (Zašto?)

```
a.stanje().dodaj(100);    ✗
```



Popravak: Funkcija koja vraća *this

```
class Racun {  
    ...  
    public:  
        Racun& stanje() const;  
    ...  
};
```

```
Racun &Racun::stanje() const {  
    cout << "Stanje (" << id << ") = "  
        << saldo << " EUR." << endl;  
    return *this;  
}
```

- ▶ no, za `Racun a("abc");` i dalje ne radi (Zašto?)

```
a.stanje().dodaj(100);    ✗
```



Preopterećivanje pomoću `const`

- ▶ problem je što `const` verzija funkcije `stanje` vraća referencu na `const` (\Rightarrow na tome ne možemo pozvati funkciju `dodaj`)
- ▶ preopteretimo funkciju `dodaj` - `const` verzija za `const` objekte, a `non-const` verzija bolje odgovara za `non-const` objekte
- ▶ dodali parametar da možemo iskoristiti funkciju i za npr. ispis u datoteku, te pomoćnu funkciju (da ne ponavljamo kod)

```
class Racun {  
    ...  
    public:  
    ...  
    Racun& stanje(std::ostream &);  
    const Racun& stanje(std::ostream &) const;  
    private:  
    void ispis_stanja (std::ostream &os) const {  
        os << "Stanje (" << id << ") = "  
        << saldo << " EUR." << std::endl;  
    }  
    ...  
};
```



Preopterećivanje pomoću `const` (nastavak)

```
Racun &Racun::stanje(std::ostream &os) {  
    ispis_stanja(os);  
    return *this;  
}  
  
const Racun &Racun::stanje(std::ostream &os) const  
{  
    ispis_stanja(os);  
    return *this;  
}
```

```
Racun a("abc");  
const Racun b("def", 10, 500);  
a.stanje(cout).dodaj(100).stanje(cout);  
b.stanje(cout);
```



Klasa Korisnik

- ▶ klasa koja će pamtit i korisnikov id (npr. OIB), ime, prezime i sve njegove račune
- ▶ dodamo u Racuni.h:

```
class Korisnik {
public:
    Korisnik(const std::string &o, const
              std::string &i, const std::string &p,
              const Racun &r) : oib(o), ime(i),
                               prezime(p) {
        racuni.push_back(r);
    }
private:
    std::string oib, ime, prezime;
    std::vector<Racun> racuni;
};
```

Klasa Korisnik - funkcija za ispis računa korisnika

```
class Korisnik {
public:
    Korisnik &info(std::ostream &os) {
        os << "Racuni korisnika " << oib
           << ":" << std::endl;
        for(auto r : racuni)
            r.ispis_stanja(os);    //!
        return *this;
    }
    ...
};
```

Problem: info treba pristup privatnim dijelovima klase Racun

Klase friendovi

Rješenje: klasa `Racun` odredi klasu `Korisnik` kao frienda

```
class Racun {  
    friend class Korisnik;  
    ...  
};
```

Primjer. U funkciji `main`:

```
Korisnik a("12345678", "John", "Doe", Racun("123"));  
a.info(cout);
```

Ispis:

```
Racuni korisnika 12345678:  
Stanje (123) = -3.98 EUR.
```



Funkcije članice kao friendovi

- ▶ uočimo da nam nije trebala cijela klasa `Korisnik` kao friend nego samo njena funkcija članica `info`
- ▶ prema tome je umjesto prethodnog u našem primjeru dovoljno:

```
class Racun {  
    friend Korisnik &Korisnik::info(std::ostream &);  
    ...  
};
```

- ▶ tu dolazi do problema pri kompajliranju - treba reorganizirati kod (npr. kompajler nailaskom na tu liniju ne zna da tek kasnije deklariramo `Korisnik`)



Reorganizacija koda

- ▶ klasa `Korisnik` treba znati da postoji `Racun`, a onda `Racun` kasnije dobije definiciju od `Korisnik` (definicije `Korisnik::Korisnik` i `Korisnik::info` prebacili u `Racun.cpp` → idući slajd):

```
class Racun;
```

```
class Korisnik {  
public:  
    Korisnik(const std::string &, const std::string &,  
             const std::string &, const Racun &);  
    Korisnik &info(std::ostream &);  
    ...  
};  
  
class Racun {  
    friend Korisnik &Korisnik::info(std::ostream &);  
    ...  
};
```



Reorganizacija koda (nastavak)

```
Korisnik::Korisnik(const std::string &o, const  
                  std::string &i, const std::string &p,  
                  const Racun &r) : oib(o), ime(i), prezime(p)  
{  
    racuni.push_back(r);  
}  
  
Korisnik &Korisnik::info(std::ostream &os) {  
    os << "Racuni korisnika " << oib  
      << ":" << std::endl;  
    for(auto r : racuni)  
        r.ispis_stanja(os);  
    return *this;  
}
```



Dodavanje tečaja

- ▶ želimo ispisati ne samo stanje u EUR nego i u CHF - npr.

Stanje (123) = 1000 EUR (= 990 CHF) .

- ▶ dopunimo klasu `Racun` i funkciju `Racun::ispis_stanja` (nju poziva funkcija `Racun::dodaj`)

```
class Racun {
    ...
private:
    valuta tecaj = 0.99;
    void ispis_stanja (std::ostream &os) const {
        os << "Stanje (" << id << ") = "
           << saldo << " EUR ( = " << saldo * tecaj
           << " CHF) ." << std::endl;
    }
}
```

```
Racun a("123",1000,1500);
a.stanje(cout);
```



Static članovi klase

- ▶ **Problem:** Ako se tečaj promijeni, treba promijeniti podatke u svim klasama.

⇒ tečaj pripada klasi, a ne pojedinom objektu klase!

```
class Racun {
    ...
public:
    static void postavi_tecaj(double);
    ...
private:
    static double tecaj;
    ...
}
```



Static članovi postoje izvan objekta

- ▶ objekti ne sadrže podatke koji se na njih odnose
- ▶ `static` funkcije članice ne dobivaju implicitno `this` pokazivač

Primjer.

```
static void postavi_tecaj(double) const;      X  
void Racun::postavi_tecaj(double t) {  
    tecaj = t;  
    id = "abc";      X  
}
```



Potrebne definicije

- ▶ za `static` podatke definicija izvan klase (samo jednom \Rightarrow zbog problema s linkerom u `Racun.cpp`)
- ▶ uočite: za funkciju nismo morali opet staviti riječ `static`

```
double Racun::tecaj = 0.99;  
  
void Racun::postavi_tecaj(double t) {  
    tecaj = t;  
}
```

Primjer. U funkciji `main`:

```
Racun a, b;  
b.stanje(cout);  
Racun::postavi_tecaj(0.95);  
a.stanje(cout);  
a.postavi_tecaj(0.96);  
b.stanje(cout);
```

