

Kontrola kopiranja: **Kopiranje**

Objektno programiranje - 6. vježbe (1. dio)

Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

12. travnja 2023. godine



Kontrola kopiranja: Pregled sadržaja

- ▶ specificiramo što se događa kad objekt nekog tipa klase: kopiramo, premještamo, pridružimo mu objekt istog tipa klase (kopiranjem ili premještanjem), uništimo
- ▶ pet posebnih funkcija članica za tu kontrolu (postoje i *defaultne*):
 - ▶ *copy* konstruktor
 - ▶ *copy-assignment* operator
 - ▶ *move* konstruktor
 - ▶ *move-assignment* operator
 - ▶ destruktor
- ▶ o *move*-ovima više u sljedećoj prezentaciji



Defaultni, sintetizirani *copy* konstruktor

Podsjetnik - koje smo konstruktore imali za `Racun`:

```
class Racun {  
    ...  
    Racun() = default;  
    Racun(const std::string &, valuta, valuta);  
    Racun(const std::string &);  
    Racun(std::istream &);  
    ...  
}
```

- ▶ čak i ako imamo te konstruktore, svejedno se sintetizirao *copy* konstruktor koji kopira članove član po član:

```
Racun a("ABC", 150, 235);  
a.dodaj(100);  
Racun b(a);      ✓  
ispis(cout, a) << endl;  
ispis(cout, b) << endl;
```



Kako bi taj konstruktor izgledao da smo ga mi pisali

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

```
Racun::Racun(const Racun &r) :  
    br_transakcija(r.br_transakcija),  
    id(r.id),  
    saldo(r.saldo),  
    prekoracenje(r.prekoracenje) { };
```

- ▶ da se ne bi („bekonačno”) rekurzivno pozivao, mora primiti **referencu** na `Racun` (dok je `const` opcionalan)
- ▶ uočimo: `b` s prethodnog slajda ima `br_transakcija` jednak 1 (iako nije sudjelovao u onome što smatramo transakcijom)



Primjer 1. Kada se zove *copy* konstruktor?

- ▶ ako u tijelo *copy* konstruktora s prijašnjeg slajda dodamo:

```
cout << "copy const!" << endl;
```

što se ispiše prilikom izvođenja sljedećeg koda? Obrazložite!

```
Racun f1(const Racun a) {
    a.stanje(cout);
    return a;
}

int main() {
    Racun a("ABC", 150, 235);
    Racun b = a; //copy inicijalizacija
    f1(b);
    return 0;
}
```



Primjer 2. Kada se zove *copy* konstruktor?

- ▶ isto pitanje kao na prethodnom slajdu, ali za sljedeći kod:

```
const Racun& f1(const Racun &a) {
    a.stanje(cout);
    return a;
}

int main() {
    vector<Racun> v{Racun("abc"), Racun("123")};
    for(auto &r : v)
        f1(r);
    return 0;
}
```



Copy-assignment operator

- ▶ **operator pridruživanja** - funkcija `operator=`
- ▶ jedan od **preopterećenih operatora** (funkcija čije je ime "operator" + simbol za taj operator)
- ▶ kao kod *copy* konstruktora dobivamo ga sintetiziranog:

```
Racun a, b("abc");  
a = b;    ✓
```

- ▶ kako bi izgledao kad bi ga mi pisali:

```
Racun {  
    ...  
    Racun& operator=(const Racun &r);  
    ...  
}
```

Komentar: uočite okvireno - želimo moći npr. `a = b = c;`



Nastavak koda s prethodnog slajda

```
Racun& Racun::operator=(const Racun &r) {  
    br_transakcija = r.br_transakcija;  
    id = r.id;  
    saldo = r.saldo;  
    prekoracenje = r.prekoracenje;  
    return *this;  
}
```



Destruktor (što želimo nakon zadnje upotrebe objekta)

- ▶ za oslobađanje resursa koje objekt koristi
- ▶ nema povratnih vrijednost
- ▶ nema parametara
 - ⇒ ne može se preopteretiti (⇒ svaka klasa ima samo jedan destr.)

Primjer. Destruktor za našu klasu kad bi ga sami pisali:

```
class Racun {  
    ...  
    ~Racun() { };  
    ...  
}
```

- ▶ prvo se izvršava tijelo destruktora, a onda se članovi „unište” (u obratnom poretku od inicijalizacije)

Važno: implicitna destrukcija člana koji je nekog od **ugrađenih** tipova **pokazivača** ne radi destrukciju objekta na koji pokazuje!

Primjer. Odredite što se ispiše: (main → sljedeći slajd)

```
class Prva {  
    std::string ime;  
public:  
    Prva(std::string str) : ime(str) { }  
    ~Prva() {  
        std::cout << "Destr1: " << ime << std::endl;  
    }  
};  
  
class Druga {  
    std::string ime;  
    Prva prva1, prva2;  
public:  
    Druga(std::string str) :  
        ime(str), prva1(str), prva2(str+"1") { }  
    ~Druga() {  
        std::cout << "Destr2: " << ime << std::endl;  
    }  
};
```

Funkcija `main` uz prethodni slajd

```
using namespace std;

int main() {
    Druga a("a");
    {
        Druga b("b");
    }
    vector<Druga> v{Druga("c")};
    cout << "Neki tekst." << endl;
    return 0;
}
```



Primjer. Klasa koja uz destruktor treba i kop. i pridr.

```
struct A {
    int* br;
    A() : br(new int(5)) { }
    //A(const A &x) { br = new int(5); }
    ~A() { delete br; }
};
```

```
void f(const A a) {
    cout << *(a.br)
         << endl;
}
```

```
int main() {
    A a;
    f(a);
    return 0;
}
```

- ▶ ako ostavimo zakomentirano **copy konstruktor**, dobiva se nešto poput:

```
5
free(): double free
detected in tcache 2
Aborted (core dumped)
```

- ▶ naravno, kako bi radilo i npr.

```
A a, b; f(a = b);
```

trebamo i svoj **operator**



Primjer. Klasa koja treba kop. i pridr. ali ne i destruktor

- ▶ želimo da svaki objekt klase ima svoj jedinstveni broj `id`

```
struct A {  
    static int brojac;  
    int podatak, id;  
    A(int b) : podatak(b), id(brojac++) { }  
    A(const A &a) : podatak(a.podatak),  
                  id(brojac++) {}  
    A& operator=(const A& a) {  
        podatak = a.podatak;  
        return *this;  
    }  
};  
  
int A::brojac = 0;
```



Nastavak primjera

```
void ispis(A a) {  
    cout << a.podatak << ", " << a.id << endl;  
}  
  
int main() {  
    A a(500), b(600);  
    ispis(a = b);  
    return 0;  
}
```

Pitanje. Što prethodni program ispiše?

- ▶ ako trebamo vlastiti konstruktor kopiranjem, tada vjerojatno trebamo i vlastiti operator pridruživanja kopiranjem



Upotreba = default

- ▶ eksplicitno tražimo kompajler da generira sintetizirane verzije članica klase za kontrolu kopiranja
- ▶ ako ne tražimo da budu *inline*, onda = default navodimo pri definiciji članice

```
class Racun {  
    ...  
    Racun(const Racun &) = default;  
    Racun& operator=(const Racun &);  
    ~Racun() = default;  
    ...  
};
```

Racun.h

```
Racun& Racun::operator=(const Racun &) = default;
```

Racun.cpp



Definiranje funkcija kao obrisanih

- ▶ obrisana funkcija je ona koja je deklarirana (pomoću = delete), ali se ne može koristiti
- ⇒ možemo zabraniti kopiranje i pridruživanje kopiranjem
- ▶ destruktor nije poželjno staviti kao obrisan

Primjer.

```
struct A {  
    int br;  
    A() = default;  
    A(const A&) = delete;  
    A &operator=(const A&) = delete;  
    ~A() = default;  
};
```

```
A a, b(a), c=a, d;    X  
a.br = 1;  
d = a;    X
```



Članovi za kontrolu kopiranja sintetizirani kao obrisani

- ▶ ako član klase ne može biti defaultno konstruiran, kopiran, pridružen ili uništen, tada je pripadna članica te klase obrisana
- ▶ članovi za kontrolu kopiranja su sintetizirani kao obrisani ako nije moguće kopirati, pridružiti ili uništiti član klase

Primjer 1.

```
struct A {  
    int &br;  
};
```

`A a;` **X**

Primjer 2.

```
struct A {  
    int &br;  
    A(int x) : br(x) { }  
};
```

```
int x = 5;  
A a(x), b(x);  
a = b;     X
```



Kontrola kopiranja i upravljanje resursima...

...koji se ne nalaze u klasi (⇒ treba destruktor ⇒ treba kontrola kopiranja)

(1.) Klase koje se ponašaju poput vrijednosti

- ▶ imaju kopiju nezavisnu od originala

Primjer. (Vektori intova (1.))

```
class Vektor {  
public:  
    Vektor(int);     //prima br. elemenata (dim)  
    Vektor(const Vektor &);  
    Vektor& operator=(const Vektor&);  
    ~Vektor();  
    Vektor& ispis(ostream &);  
    Vektor& unos(istream &);  
private:  
    int dim, *elementi;  
};
```



Nastavak primjera (funkcije za unos i ispis vektora)

```
Vektor& Vektor::unos(istream &is) {  
    for(int i = 0; i < dim; ++i)  
        is >> elementi[i];  
    return *this;  
}
```

```
Vektor& Vektor::ispis(ostream &os) {  
    os << "Dim " << dim << ": ";  
    for(int i = 0; i < dim; ++i)  
        os << elementi[i] << " ";  
    os << endl;  
    return *this;  
}
```



Nastavak primjera (konstruktor, *copy* kon. i destruktor)

```
Vektor::Vektor(int d = 0) { //defaultni parametar!  
    dim = d;  
    elementi = new int[d];  
}
```

```
Vektor::Vektor(const Vektor &v) {  
    dim = v.dim;  
    elementi = new int[dim];  
}
```

```
Vektor::~Vektor() {  
    delete[] elementi;  
}
```



Nastavak primjera - **operator=**

- ▶ doima se da bi ovo bila ispravna implementacija:

```
Vektor& Vektor::operator=(const Vektor &v) {  
    delete[] elementi;  
    dim = v.dim;  
    elementi = new int[v.dim];  
    for(int i = 0; i < v.dim; ++i)  
        temp[i] = v.elementi[i];  
    return *this;  
}
```

- ▶ objasnite koji se problem javlja pri izvršavanju sljedećeg koda:

```
Vektor v, w2(5);  
v.ispis(cout);  
cout << "Unesite 5 elemenata: ";  
w2.unos(cin).ispis(cout);  
v = v = w2;  
v.ispis(cout);
```



Popravak prethodnog koda

```
Vektor& Vektor::operator=(const Vektor &v) {  
    int *temp = new int[v.dim];  
    for(int i = 0; i < v.dim; ++i)  
        temp[i] = v.elementi[i];  
    delete[] elementi;  
    dim = v.dim;  
    elementi = temp;  
    return *this;  
}
```

Unos/ispis za kod s prethodnog slajda korištenjem gornjeg operatora:

```
Dim 0:  
Unesite 5 elemenata: 1 3 2 4 5  
Dim 5: 1 3 2 4 5  
Dim 5: 1 3 2 4 5
```



(2.) Klase koje se ponašaju poput pokazivača

- ▶ kopiramo pokazivače, a ne resurs na koji pokazuju
 - ▶ destruktor smije osloboditi resurse na koje pokazivači pokazuju tek kad **uništavamo posljednji pokazivač** na njih
- ⇒ brojimo reference na resurs (dinamički alociran brojač, a ne dio objekta - želimo da uvijek pokazuje stvarno stanje)

Primjer. (Vektori intova (2.))

```
class Vektor {
public:
    Vektor(int);    //prima br. elemenata (dim)
    Vektor(const Vektor &);
    Vektor& operator=(const Vektor&);
    ~Vektor();
    Vektor& ispis(ostream &);
    Vektor& unos(istream &);
private:
    int dim, *elementi, *br_ref;
};
```



Nastavak primjera: (*copy*) konstruktor

- ▶ funkcije `Vektor::ispis` i `Vektor::unos` su kao i prije
- ▶ konstruktor alocira novi resurs i brojač koji postavlja na 1

```
Vektor::Vektor(int d = 0) :
    dim(d),
    elementi(new int[d]),
    br_ref(new int(0)) { }
```

- ▶ *copy* konstruktor kopira podatke i poveća brojač

```
Vektor::Vektor(const Vektor &v) :
    dim(v.dim),
    elementi(v.elementi),
    br_ref(v.br_ref) {
    ++(*br_ref);
}
```



Nastavak primjera: destruktor

- ▶ ne možemo usloboditi resurs ako još ima objekata koji imaju pokazivač na njega

```
Vektor::~~Vektor() {  
    --(*br_ref);  
    if(*br_ref == 0) {  
        delete[] elementi;  
        delete br_ref;  
    }  
}
```

- ▶ uočimo: prije gornjeg `if`-a treba dekrementirati brojač!



Nastavak primjera: operator pridruživanja kopiranjem

- ▶ povećamo brojač desnog operanda i smanjimo brojač lijevog operanda (uz oslobađanje resursa ako je potrebno)

```
Vektor& Vektor::operator=(const Vektor &v) {  
    ++(*v.br_ref);  
    --*br_ref;  
    if(*br_ref == 0) {  
        delete[] elementi;  
        delete br_ref;  
    }  
    dim = v.dim;  
    elementi = v.elementi;  
    br_ref = v.br_ref;  
    return *this;  
}
```

Pitanje. Zašto je važno odmah prvo napraviti `++(*v.br_ref)`?



Primjer `main` funkcije za ilustraciju prethodnog koda

```
int main() {  
    Vektor v, w2(5);  
    v.ispis(cout);  
    cout << "Unesite 5 elemenata: ";  
    w2.unos(cin).ispis(cout);  
    v = v = w2;  
    v.ispis(cout);  
    return 0;  
}
```

- ▶ analizirati što se događa tijekom izvršavanja gornjeg koda

Definiranje vlastite `swap` funkcije

- ▶ važno algoritmima koji mijenjaju redoslijed elemenata

Primjer. Standardni `swap` bi ovako zamijenio dva vektora `v1` i `v2`:

```
Vektor temp = v1;  
v1 = v2;  
v2 = temp;
```

- ▶ promatramo vektore intova, implementaciju (1.)
- ⇒ Koliko ovdje ima (nepotrebnih) kopiranja memorije?
- ▶ možemo napisati `swap` funkciju specifičnu za našu klasu koja bi se onda pozivala umjesto standardne
- ▶ bit će optimalnija jer će samo zamijeniti pokazivače

swap funkcija za vektore (implementacija (1.))

- ▶ `inline` za optimizaciju (Zašto mora biti `friend`?)
- ▶ zbog optimizacije **općenito** želimo da se zovu specifične `swap` funkcije gdje mogu (umjesto standardne `std::swap` funkcije)

```
class Vektor {  
    friend void swap(Vektor&, Vektor&);  
    ...  
};  
  
inline void swap(Vektor &a, Vektor &b) {  
    using std::swap;  
    swap(a.dim, b.dim);  
    swap(a.elementi, b.elementi);  
}
```

```
Vektor v(4), w(5);  
...  
swap(v, w);
```



Zadatak

- ▶ potrebno je implementirati klase `Datoteka` i `Mapa`
- ▶ svaka `Datoteka` ima dva podatka: `string` u kojem je spremljen sadržaj datoteke i skup pokazivača na mape u kojima se ta datoteka nalazi (`set<Mapa*>`)
- ▶ svaka `Mapa` sadrži skup pokazivača na datoteke koje se u njoj nalaze (`set<Datoteka*>`)
- ▶ obje klase imaju svoje metode `dodaj` i `ukloni` za dodavanje datoteke u mapu
- ▶ razmisliti što moramo sve ažurirati pri korištenju operatora `=` i `copy` konstruktora
- ▶ pritom, ukoliko kopiramo datoteku, dobivamo dvije različite datoteke koje se moraju javljati u istim mapama
- ▶ dodatno definirati i svoju funkciju

```
void swap(Datoteka&, Datoteka&);
```



Copy elision (zaobilaženje kopiranja)

- ▶ optimizacija koju implementira **većina** kompajlera kako bi se spriječila (možda skupa i nepotrebna!) kopiranja ([više info](#))

Primjer. Dodajmo klasi `Vektor` (prijateljsku) funkciju `povecaj`:

```
class Vektor {
    friend Vektor povecaj(const Vektor&);
    ...
}

Vektor povecaj(const Vektor &v) {
    Vektor rez(v.dim);
    for(int i = 0; i < v.dim; ++i)
        rez.elementi[i] = v.elementi[i] + 1;
    return rez;
}
```



Nastavak primjera

- ▶ Dodajmo svim funkcijama koje imamo poruku koja se ispiše na početku izvršavanja tih funkcija, npr. za `Vektor::~Vektor`:

```
Vektor::~~Vektor() {
    cout << "Destruktor" << endl;
    delete[] elementi;
}
```

- ▶ tada za sljedeći kod (i unos 1 3 2 4) možemo dobiti ispis desno:

```
int main() {
    Vektor v(4);
    v.unos(cin);
    Vektor w = povecaj(v);
    w.ispis(cout);
    return 0;
}
```

Pitanje: Jesmo li dobili očekivano?

```
Konstruktor
Unos funkcija
1 3 2 4
Povecaj funkcija
Konstruktor
Ispis funkcija
Dim 4:  2 4 3 5
Destruktor
Destruktor
```

