Kontrola kopiranja: Premještanje

Objektno programiranje - 7. vježbe (2. dio)

dr. sc. Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

10. travnja 2024. godine

Podsjetnik na klasu `Vektor` iz prošle prezentacije

```
class Vektor {
    friend void swap(Vektor&, Vektor&);
    friend Vektor povecaj(const Vektor&);
    public:
        Vektor(int);
        Vektor(const Vektor &);
        Vektor& operator=(const Vektor&);
        ~Vektor();
        Vektor& ispis(ostream &);
        Vektor& unos(istream &);
    private:
        int dim, *elementi;
};
```

Premještanje: Motivacija

- Ako svaka funkcija koju imamo ispiše prigodnu poruku, ispis za sljedeći kod mogao bi biti ovaj prikazan desno:

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

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

Premještanje: Motivacija (nastavak)

Podsjetnik na operator pridruživanja kopiranjem (iz prethodne prez.):

```
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;
}
```

Dio koda s prethodnog slajda:

```
w = povecaj(v);
```

- napravili kopiju objekta kojeg odmah zatim uništili (**desna vrijednost**)

$\Rightarrow$ premještanje umjesto kopiranja daje bolje performanse

Move konstruktor (Konstruktor premještanjem)

- umjesto kopiranja resursa, **preuzmemo ih**
- za razliku od *copy*-konstruktora, parametar je **desna referenca**
- signalizirali konstruktoru da ne može izazvati iznimku

```
class Vektor {
    ...
  public:
    Vektor(Vektor&&) noexcept;
    ...
};

Vektor::Vektor(Vektor &&v) noexcept {
    dim = v.dim;
    elementi = v.elementi;
    v.elementi = nullptr;
}
```

Konstruktor premještanjem

- možemo ga eksplicitno pozvati korištenjem **`std::move`**
- objekt čije resurse uzeli treba ostaviti u stanju u kojem se može uništiti ili mu se može pridružiti nova vrijednost
- ako i *move* konstruktoru dodamo ispis poruke `Move konstruktor`:

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

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

Pitanje. Što bi pošlo po krivu ako bi iz konstruktora s prethodnog slajda uklonili `v.elementi = nullptr;`?

Operator pridruživanja premještanjem

```
class Vektor {
    ...
  public:
    Vektor& operator=(Vektor &&) noexcept;
    ...
};

Vektor& Vektor::operator=(Vektor &&v) noexcept {
    if (this != &v) {    //provjera protiv samopridruživanja
        delete[] elementi; //oslobodimo postojeće resurse
        dim = v.dim;
        elementi = v.elementi; //preuzmemo resurse od v
        v.elementi = nullptr; //ostavimo v u destrukt. stanju
    }
    return *this;
}
```

Podsjetnik. Zašto je važna provjera protiv samopridruživanja?

Prikaz na primjeru s početka ove prezentacije

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

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

- ako klasa definira svoj *copy*-konstr. ili op. pridr. kopiranjem ili destruktor, *move* konstruktor i operator pridruživanja premještanjem se ne sintetiziraju

$\Rightarrow$ prošli put bio pozvan operator pridruživanja kopiranjem

Kad se sintetizira *defaultni move* konstr. i operator =

Uvjeti:

(1.) ne smijemo imati svoju *copy* kontrolu (konstr/destr/op=)

(2.) svaki ne-*static* član klase ima svoj *move*

```
struct A {
    int br;  //ugrađeni tipovi imaju svoj std::move
    string str;  //i stringovi imaju svoj std::move
    A() = default;
    A(const A&) = delete;
    ~A() { };
};

int main() {
    A a1;
    A a2 = std::move(a1); ✗  //zove obrisan copy konst
    return 0;
}
```

Eksplicitno zahtijevanje *defaultnog move* konstruktora

- sada je sve u redu:

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

int main() {
    A a1;
    A a2 = std::move(a1);  ✓
    return 0;
}
```

Zadatak

Napisati konstruktor premještanjem i operaciju pridruživanja premještanjem za klase `Datoteka` i `Mapa` (iz zadatka s prošle prezentacije).

`Move` iteratori

- zapravo su **adaptori** za iteratore
- mijenjaju ponašanje **dereferenciranja iteratora**: dobivamo desnu referencu na objekt umjesto lijeve
- ostale operacije s iteratorima odvijaju se normalno
- za pretvaranje „običnog” u `move` iterator - funkcija **`make_move_iterator`**
- primjeri na idućim slajdovima:

```
vector<Vektor> w(v.begin(), v.end());
```

vs.

```
vector<Vektor> w(make_move_iterator(v.begin()),
                 make_move_iterator(v.end()));
```

Iako korisna, **std::move je opasna operacija** - moramo biti sigurni da će objekt kojem preuzeli resurse biti ili za destrukciju ili za pridruživanje novog resursa!

Komentirajte ispis (sa zarezom umjesto `std::endl`)

```
int main() {
    Vektor a, b, c;
    vector<Vektor> v;
    cout << v.capacity() << endl;
    v.push_back(a);
    cout << v.capacity() << endl;
    v.push_back(b);
    cout << v.capacity() << endl;
    v.push_back(c);
    vector<Vektor> w(v.begin(),v.end());
    return 0;
}
```

Konstruktor, Konstruktor, Konstruktor, 0, Copy konstruktor, 1, Copy konstruktor, Move konstruktor!, Destruktor, 2, Copy konstruktor, Move konstruktor!, Move konstruktor!, Destruktor, Destruktor, Copy konstruktor, Copy konstruktor, Copy konstruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor

Komentirajte ispis (sa zarezom umjesto `std::endl`)

```
int main() {
    Vektor a, b, c;
    vector<Vektor> v;
    v.reserve(3);
    cout << v.capacity() << endl;
    v.push_back(a);
    cout << v.capacity() << endl;
    v.push_back(b);
    cout << v.capacity() << endl;
    v.push_back(c);
    vector<Vektor> w(v.begin(),v.end());
    return 0;
}
```

Konstruktor, Konstruktor, Konstruktor, 3, Copy konstruktor, 3, Copy konstruktor, 3, Copy konstruktor, Copy konstruktor, Copy konstruktor, Copy konstruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor

Komentirajte ispis (sa zarezom umjesto `std::endl`)

```
int main() {
    Vektor a, b, c;
    vector<Vektor> v;
    v.reserve(3);
    cout << v.capacity() << endl;
    v.push_back(a);
    cout << v.capacity() << endl;
    v.push_back(b);
    cout << v.capacity() << endl;
    v.push_back(c);
    vector<Vektor> w(make_move_iterator(v.begin()),
                     make_move_iterator(v.end()));
    return 0;
}
```

Konstruktor, Konstruktor, Konstruktor, 3, Copy konstruktor, 3, Copy konstruktor, 3, Copy konstruktor, Move konstruktor!, Move konstruktor!, Move konstruktor!, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor, Destruktor