Dinamička memorija

Objektno programiranje - 8. vježbe (1. dio)

Sebastijan Horvat

Prirodoslovno-matematički fakultet,
Sveučilište u Zagrebu

10. svibnja 2023. godine

Kratko ponavljanje

Što rade sljedeće instrukcije:

(a) `int *pi = new int;`

(b) `int *pi = new int(1024);`

(c) `int *pi = new int();`

(d) `string *ps = new string;`

(e) `string *ps = new string();`

(f) `string *ps = new string(10, '9');`

(g) `vector<int> *pv = new vector<int>{0,1,2,3,4,5};`

(h) `auto p1 = new auto("abc");  //char**`

(i) `const int *pci = new const int(1024);`

(j) `const int *pci = new const int;` ✗

(k) `const string *pcs = new const string;`

Ponavljanje

- alociranje niza: `int *p = new int[42];`
- inicijalizacija: `int *p2 = new int[10]();`
  `int *p3 = new int[10]{0,1,2,3,4,5,6,7,8,9};`

Primjer. Odredite vrijednosti elemenata dinamičkih polja:

(a) `string *p = new string[10];`

(b) `string *p = new string[10]();`

(c) `string *p = new string[10]{"a","b",string(5,'x')};`

Pitanje. Zašto dealokaciju moramo raditi ovako: `delete [] p;`
a ne ovako: `delete p;`?

Pitanje. Što ako korisnik unese za `n` vrijednosti 0, a koristimo:
`string *p = new string[n];`?

- neuspjela alokacija baca iznimku `std::bad_alloc`
- to možemo suzbiti s `nothrow` (tada samo vraća null-pokazivač)
- `delete` mora dobiti ili null-pokazivač ili pokazivač na dinamički alociranu memoriju pomoću `new`

Primjer.

```
int *p1 = nullptr;
int *p2 = new (nothrow) int; //treba zaglavlje new
delete p1;
delete p2;
p1 = p2 = nullptr;
```

Pametni pokazivači: Motivacija

- dinamička memorija problematična jer ju treba osloboditi u pravom trenutku
- česti problemi: zaboravljanje oslobađanja memorije ili oslobađanje dok još neki pokazivač na nju pokazuje
- podsjetnik: *klase koje se ponašaju poput pokazivača*

Rješenje: **pametni pokazivači** (zaglavlje `memory`)

- prate koliko pokazivača imamo na neki resurs i znaju kada ga treba osloboditi
- dvije vrste (posjeduje li pokazivač resurs ili ga dijeli):
  - **shared_ptr**
  - **unique_ptr**

shared_ptr

- predložak za tip - navodimo tip objekta na koji će pokazivati
- po *defaultu* je nul-pokazivač
- funkcija `make_shared` alocira i inicijalizira (koristi argumente za konstrukciju - poput *emplace*) objekt, te vraća `shared_ptr` na njega

Primjer.

```
shared_ptr<string> p1;
shared_ptr<list<int>> p2;
shared_ptr<int> p3 = make_shared<int>(42);
shared_ptr<string> p4 = make_shared<string>(10,'9');
shared_ptr<int> p5 = make_shared<int>();
auto p6 = make_shared<vector<string>>();
```

Primjer. `shared_ptr` ima brojač referenci

```
#include<memory>
...

struct Par {
     Par(int a, int b) :  br(a), naz(b) { }
     int br, naz;
};

void ucitaj(shared_ptr<Par> p) {
     cin >> p->br >> p->naz;
}

void ispis(shared_ptr<Par> p) {
     cout << "(" << p->br << "," << p->naz
          << ") " << p.use_count() << endl;
}
```

(`main` je na sljedećem slajdu...)

Nastavak primjera (`main` funkcija)

```
int main() {
      auto p1 = make_shared<Par>(1,5);
      ispis(p1);
      {
          auto p2(p1); //ili: auto p2 = p1;
          ucitaj(p2);
          ispis(p2);
      }
      ispis(p1);
      p1 = make_shared<Par>(3,4);
      ispis(p1);
      return 0;
}
```

- kad broj referenci padne na 0, resurs se automatski oslobađa
- kad bi `Par` imao *defaultni* konstruktor imali bismo, primjerice, `p1 = make_shared<Par>();`

Din. memoriju često koristimo za dijeljenje resursa

Primjer. Ne znamo unaprijed broj stringova koje trebamo
$\Rightarrow$ koristimo `vector`

```
vector<string> v1;
{
    vector<string> v2 = {"a", "an", "the"};
    v1 = v2;
}
```

- bolje umjesto kopiranja (za velik broj elemenata neefikasno!) da imaju dijeljeni resurs
- no, sjetimo se količine posla iz dijela *klase koje se ponašaju poput pokazivača* - možemo izbjeći ovako:

```
class vektor {
 private:
  std::shared_ptr<std::vector<std::string>> podaci;
};
```

Datoteka `vektor.h`

- dostupna na web stranici kolegija (zajedno s datotekama `Vektor.cpp` i `main.cpp`)

```
class vektor {
  public:
    typedef std::vector<std::string>::size_type
            size_type; //za kraci zapis
    vektor();
    vektor(std::initializer_list<std::string>);
    size_type velicina() const;
    bool prazan() const;
    std::string& dohvati(size_type);
    void dodaj(const std::string&); //dodaje na kraj
    void izbaci(); //izbaci s kraja
  private:
    std::shared_ptr<std::vector<std::string>>
          podaci;
};
```

Konstruktori

```
vektor::vektor() :
     podaci(make_shared<vector<string>>()) { }

vektor::vektor(initializer_list<string> il) :
     podaci(make_shared<vector<string>>(il)) { }
```

- `dohvati` i `izbaci` operacije moraju provjeriti postoji li određeni element, te prijaviti grešku ako ne postoji

$\Rightarrow$ dodamo privatnu funkciju `provjeri`

```
class vektor {
  private:
     ...
     void provjeri(size_type,
                   const std::string&) const;
};
```

Nastavak (metode `dohvati` i `izbaci`)

```
void vektor::provjeri(size_type i,
                      const string &poruka) const {
     if (i >= podaci->size())
          throw out_of_range(poruka);
}
```

Upotreba u funkcijama `dohvati` i `izbaci`:

```
string& vektor::dohvati(size_type i) {
     provjeri(i, "nema trazenog elementa");
     return (*podaci)[i];
}

void vektor::izbaci() {
     provjeri(0, "izbacivanje iz praznog vektora");
     podaci->pop_back();
}
```

Nastavak (metode `velicina`, `prazan`, `dodaj`)

- dovoljno jednostavne za *inline*

```
class vektor {
    public:
        ...
        size_type velicina() const {
            return podaci->size();
        }
        bool prazan() const {
            return podaci->empty();
        }
        ...
        void dodaj(const std::string &s) {
            podaci->push_back(s);
        }
        ...
};
```

Primjer `main` funkcije

```
vektor v({"jedan","dva","tri"});
{
    vektor v1 = v;
    v1.izbaci();
    v1.dodaj("pet");
}
for(size_t i = 0; i < v.velicina(); ++i)
    cout << v.dohvati(i) << endl;
```

Obrazložite zašto su nam sad dobre *defaultne* verzije operacija kopiranja, pridruživanja, te destruktora.

- ne možemo implicitno pretvoriti pokazivač u pametni pokazivač

Primjer.

```
shared_ptr<int> p2(new int(42));    ✓
shared_ptr<int> p1 = new int(1024);    ✗

shared_ptr<int> clone(int p) {
    return new int(p);    ✗
    return shared_ptr<int>(new int(p));    ✓
}
```

- nije preporučljivo miješati pametne pokazivače s običnim pokazivačima (samo pametni znaju kad treba osloboditi memoriju)

shared_ptr operacije

- funkcija **get** vraća (običan) pokazivač na objekt na koji pokazuje
- svrha: ako nam treba običan umjesto pametnog pokazivača (oprez: ne preko običnog pokazivača uništiti objekt!)

Primjer. Objasnite što je ovdje pošlo po krivu:

```
shared_ptr<int> p(new int(42));
int *q = p.get();
{
     shared_ptr<int>(q);
}
int br = *p;    ✗
```

- **reset** za pridruživanje novog pokazivača, a **unique** za provjeru je li to jedini pametni pokazivač na resurs

```
if (!p.unique())
    p.reset(new string(*p));
```

Korištenje vlastitog *delete* koda

- `shared_ptr` po *deafultu* pri uništenju resursa poziva `delete` na pokazivaču koji sadrži
- umjesto toga, može se pozvati funkcija koju damo pametnom pokazivaču

```
void kraj(int *p) {
    cout << "Kraj:  " << *p << endl;
    delete p;
}

...
shared_ptr<int> p(new int(5), kraj);
{
    shared_ptr<int> q(p);
}
```

unique_ptr

- „posjeduje” objekt na koji pokazuje
- nema kopiranja/pridruživanja
- nema `make_shared` $\Rightarrow$ koristimo `new`

Primjer.

```
unique_ptr<string> p1(new string("abc"));
unique_ptr<string> p2(p1);    ✗
unique_ptr<string> p3;
p3 = p1;    ✗
```

- možemo prenijeti vlasništvo jednog (nekonstantnog) `unique_ptr` drugome korištenjem `release` ili `reset`

Primjer.

```
unique_ptr<string> p2(p1.release());
unique_ptr<string> p3(new string("abc"));
p2.reset(p3.release());
```

Nema kopiranja ili pridruživanja, ali premještanje može

Primjer.

```
unique_ptr<int> f1(int p) {
     return unique_ptr<int>(new int(p));
}

unique_ptr<int> f2(int p) {
     unique_ptr<int> r(new int(p));
     return r;
}

...

unique_ptr<int> p1 = f1(12);
unique_ptr<int> p2 = f2(15);
cout << *p1 << ", " << *p2 << endl;
```

Korištenje vlastitog *delete* koda (`unique_ptr`)

- slično kao kod vlastitih *hash* funkcija za asoc. spremnike

Primjer.

```
void kraj(int *p) {
     cout << "Kraj:  " << *p << endl;
     delete p;
}

...

unique_ptr<int, decltype(kraj)*>
     p1(new int(50),kraj);
```

Pametni pokazivači i dinamička polja

Primjer.

```
unique_ptr<int[]> up(new int[10]); //zagrade!
for (size_t i = 0; i != 10; ++i)
    up[i] = i;
up.release(); //automatski poziva delete[]
```

Problem. `shared_ptr` ne podržava izravno dinamička polja.

⇒ trebamo vlastiti kod za oslobađanje (po def. zove samo `delete`)

⇒ nema operatora[ ] - ide preko `get()` i pokazivača ispod

Primjer.

```
void kraj(int *p) {
    delete[] p;
}
```

```
shared_ptr<int> sp(new int[10],kraj);
for (size_t i = 0; i != 10; ++i)
    *(sp.get() + i) = i;
sp.reset();
```

Lambda izraz

- prethodni kod jednostavnije uz lambda izraz:

```
shared_ptr<int> sp(new int[10],
    [](int *p) { delete[] p; });
```

Općenito o lambdama:

- objekti koji se mogu pozivati (uz funkcije, funkcijske pokazivače i klase koje preopterećuju `operator()`)
- općeniti oblik:

  [capture list](parameter list) -> return type { function body }
- lista parametara i povratni tip nisu obavezni
- unutar [ ] navodimo okolne lokalne varijable koje želimo koristiti

Klasa `allocator`: motivacija

- `new` = alokacija + konstrukcija objek(a)ta u memoriji
- `delete` = destrukcija + dealokacija objek(a)ta u memoriji

Primjer. Koliko elemenata se nepotrebno postavlja na neku vrijednost da bi im se ona zatim promijenila?

```
string *const p = new string[n];
string s, *q = p;
while (cin >> s && q != p + n)
      *q++ = s;
...
delete[] p;
```

- važniji problem: klase bez *defaultnog* konstruktora ne mogu se dinamički alocirati kao niz

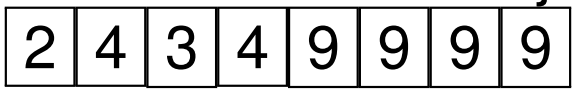
Klasa `allocator`

- `allocator` je predložak - navodimo tip objekta koji alociramo
- alocira memoriju (prikladne veličine i poravnanja) za čuvanje objekata danog tipa
- zatim možemo po potrebi konstruirati objekte (ne čitati nekonstruiranu memoriju!) - na kraju prvo uništimo sve konstruirane objekte, a zatim dealociramo svu zauzetu memoriju

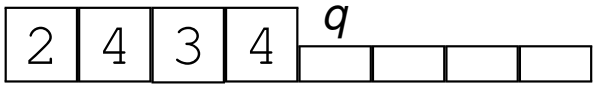
Primjer.

```
allocator<string> alloc;
auto const p = alloc.allocate(n);
auto q = p;
alloc.construct(q++);
alloc.construct(q++, 10, 'c');
alloc.construct(q++, "hi");
cout << *p << endl;
while (q != p)
    alloc.destroy(--q);
alloc.deallocate(p, n);
```

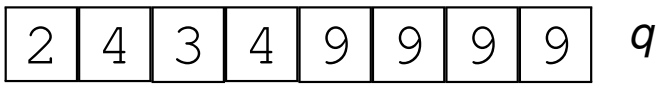
- zaglavlje **memory**

Primjer. Kopiramo elemente vektora i zatim još toliko brojeva 9 - primjerice, {2,4,3,4} $\longrightarrow$ 


```
vector<int> vi = {2,4,3,4};
allocator<int> alloc;
auto p = alloc.allocate(vi.size() * 2);
auto q = uninitialized_copy(vi.begin(),vi.end(),p);
```




```
q = uninitialized_fill_n(q, vi.size(), 9);
```




```
for(size_t i = 0; i < vi.size() * 2; ++i)
     cout << p[i] << endl;
while (q != p)
     alloc.destroy(--q);
alloc.deallocate(p, vi.size() * 2);
```

`weak_ptr` - za „slabo” dijeljenje objekta

- pametni pokazivač koji pokazuje na objekt kojim upravlja `shared_ptr`, ali ne utječe na pripadni brojač referenci

Primjer.

```
auto p = make_shared<int>(24);
weak_ptr<int> wp1(p);
{
      auto q = make_shared<int>(42);
      wp1 = q;
      cout << wp1.use_count() << endl; //1
}
cout << wp1.expired() << endl; //1
wp1.reset(); //null
wp1 = p;
if (shared_ptr<int> np = wp1.lock()) {
      cout << *np << endl; //nema deref.w.ptr!
}
```

Primjer upotrebe `weak_ptr`

Pokazivačka klasa za klasu `Vektor`

- dodamo u `vektor.h`:

```
class VektorPok {
  private:
    std::weak_ptr<std::vector<std::string>> wptr;
    std::size_t tren;
};
```

- imamo slabi pokazivač na `vector` `podaci` vektora iz kojeg ćemo ga inicijalizirati (ili `nullptr`)
- `tren` = indeks elementa koji trenutno označava ovaj objekt

`VektorPok`: konstruktori

```
class VektorPok {
    public:
        VektorPok():  tren(0) { }
        VektorPok(vektor &v, size_t i = 0) :
                wptr(v.podaci), tren(i) { }
    ...
};
```

Uočimo:

- prvi konstruktor: implicitno `wptr` kao *null* slabi pokazivač
- drugi konstruktor: uzima referencu na nekonstantni objekt tipa `vektor` $\Rightarrow$ ne možemo `VektorPok` vezati za konstantni `vektor`

VektorPok: metoda članica provjeri

- želimo provjeriti je li sigurno dereferencirati VektorPok

```
class VektorPok {
  ...
  private:
    std::shared_ptr<std::vector<std::string>>
     provjeri(std::size_t,const std::string&) const;
  ...
};
```

```
shared_ptr<vector<string>> VektorPok::provjeri
        (size_t i, const string &poruka) const {
    auto ret = wptr.lock();
    if (!ret)
        throw std::runtime_error("nema vektora");
    if (i >= ret->size())
        throw std::out_of_range(poruka);
    return ret;
}
```

Operacije s pokazivačem

```
class VektorPok {
    public:
        VektorPok& operator++();
        std::string& operator*() const;
        std::string* operator->() const;
        ...
};
```

```
VektorPok& VektorPok::operator++() {
    provjeri(tren, "inkr.  preko enda");
    ++tren;
    return *this;
}
string& VektorPok::operator*() const {
    auto p = provjeri(tren, "deref.  nakon enda");
    return (*p)[tren];
}
string* VektorPok::operator->() const {
    return & this->operator*();
}
```

Klasa `vektor`: `begin` i `end` operacije

- vraćaju `VektorPok` objekt koji pokazuju na prvi i jedan iza zadnjeg elementa u `vektor` objektu

```
class VektorPok;
class vektor {
    friend class VektorPok;
    public:
        VektorPok begin();
        VektorPok end();
        ...
};
```

```
VektorPok vektor::begin() {
    return VektorPok(*this);
}
VektorPok vektor::end() {
    return VektorPok(*this, podaci->size());
}
```

Dodatno: Operacije usporedbe za `VektorPok`

```
class VektorPok {
      friend bool operator==(const VektorPok&,
                  const VektorPok&);
      friend bool operator!=(const VektorPok&,
                  const VektorPok&);
      ...
};
```

```
bool operator==(const VektorPok &lv,
                  const VektorPok &dv) {
    return (*(lv.wptr.lock()) == *(dv.wptr.lock())
              && lv.tren == dv.tren);
}
bool operator!=(const VektorPok &lv,
                  const VektorPok &dv) {
    return !(lv == dv);
}
```

Primjer upotrebe (u funkciji `main`)

```
VektorPok p;
for(p = v.begin(); p != v.end(); ++p)
    cout << *p << endl;
```

Program za pretraživanje teksta

Zadatak. Napisati program koji u zadanoj datoteci nalazi traženu riječ (ukoliko se onda javlja u toj datoteci). Nakon unosa riječi, treba se ispisati u koliko se linija datoteka ta riječ javlja, te lista linija iz datoteke (zajedno s pripadnim rednim brojem) u kojima se ta riječ javlja. Ako se riječ javlja u istoj liniji više od jednom, ta linija se samo jednom ispiše. Linije u kojima se javlja tražena riječ treba ispisati u uzlaznom poretku, tj. 7. linija mora se ispisati prije 9. linija itd.

Primjer unosa i ispisa

```
Unesite trazenu rijec ili k za kraj:  et
et se javlja u 12 linija
     (linija 4) invidunt ut labore et dolore ...
     (linija 6) At vero eos et accusam et justo ...
     (linija 10) sed diam ...  labore et ...
     (linija 12) At vero eos et accusam et ...
     (linija 16) sed diam ...  labore et ...
     (linija 18) At vero eos et accusam et ...
     (linija 24) at vero eros et accumsan et ...
     (linija 37) at vero eros et accumsan et ...
     (linija 52) At vero eos et accusam et ...
     (linija 56) sed diam ...  labore et ...
     (linija 57) sed diam ...  vero eos et ...
     (linija 61) et nonumy ...  et et invidunt ...
Unesite trazenu rijec ili k za kraj:  dian
dian se javlja u 0 linija
```

(**Napomena**. U rješenju se umjesto ... ispiše cijela linija!)

Rješenje. (1. dio - klasa `Tekst`)

`Tekst.h`

```
using br_linije = std::vector<std::string>::size_type;

class Tekst {
 private:
  std::shared_ptr<std::vector<std::string>> datoteka;
  std::map<std::string,
           std::shared_ptr<std::set<br_linije>>> podaci;
};
```

- spremamo datoteku kao niz linija - ako kasnije trebamo liniju br. `i`, naći ćemo ju na indeksu `i`
- svakoj riječi pridružimo skup rednih brojeva linija u kojima se ta riječ javlja (Koja je ovdje prednost korištenja skupa?)
- ne želimo kopiranje podataka - no, nije poželjno da, primjerice, destruktor uništi resurs na koji imamo pokazivač
  ($\Rightarrow$ to se neće dogoditi korištenjem `shared_ptr`!)

Rješenje. (1. dio - klasa `Tekst`)

`Tekst.h`

```
class RezultatUpita;
class Tekst {
  public:
    Tekst(std::ifstream&);
    RezultatUpita upit(const std::string&) const;
  private:
    ...
};
```

- konstruktor čita danu datoteku (sprema linije i podatke o riječima)
- `upit` za traženu riječ vraća: broj linija, brojeve i sadržaj linija u kojima se ta riječ javlja - najlakše sve to vratiti u drugoj klasi `RezultatUpita` (uočite potrebnu deklaraciju!)

Klasa `Tekst` - Konstruktor

```
Tekst::Tekst(ifstream &is) :
                datoteka(new vector<string>) {
    string str;
    while (getline(is, str)) {
        datoteka->push_back(str);
        int n = datoteka->size() - 1;
        istringstream linija(str);
        string rijec;
        while (linija >> rijec) {
            auto &linije = podaci[rijec];
            if (!linije)
                linije.reset(new set<br_linije>);
            linije->insert(n);
        }
    }
}
```

Rješenje (2. dio - klasa `RezultatUpita`)

`Tekst.h`

```
class RezultatUpita {
  private:
    std::string trazeno;
    std::shared_ptr<std::set<br_linije>> linije;
    std::shared_ptr<std::vector<std::string>> datoteka;
};
```

Podaci u rezultatu upita:

- riječ koju ovaj rezultat predstavlja
- pametni pokazivač na skup brojeva linija u kojima se ta riječ javlja
- pametni pokazivač na vektor koji sadrži ulaznu datoteku

Rješenje (2. dio - klasa `RezultatUpita`)

`Tekst.h`

```
class RezultatUpita {
  public:
    RezultatUpita(std::string s,
        std::shared_ptr<std::set<br_linije>> p,
        std::shared_ptr<std::vector<std::string>> f):
            trazeno(s), linije(p), datoteka(f) { }
    ...
};
```

- konstruktor - sprema argumente u odgovarajuće dijelove

Klasa `Tekst` - funkcija `upit`

```
RezultatUpita Tekst::upit(const string &rijec)const{
    static shared_ptr<set<br_linije>>
                nemapodataka(new set<br_linije>);
    auto lokacija = podaci.find(rijec);
    if (lokacija == podaci.end())
        return RezultatUpita(rijec,
                    nemapodataka, datoteka);
    //else
        return RezultatUpita(rijec,
                    lokacija->second, datoteka);
}
```

Funkcija za ispis rezultata upita

```
class RezultatUpita {
    friend std::ostream& ispis(std::ostream&,
                const RezultatUpita&);
    ...
};
```

```
ostream &ispis(ostream &os,const RezultatUpita &rez){
    os << rez.trazeno << " se javlja u "
       << rez.linije->size() << " linij";
    switch(rez.linije->size()) {
       case 1 :  os << "i"; break;
       case 2 : case 3 : case 4 : os << "e"; break;
       default:  os << "a";
    }
    os << endl;
    for (auto br :  *rez.linije)
       os << "\t(linija " << br + 1 << ") "
          << *(rez.datoteka->begin() + br) << endl;
    return os; }
```

Primjer `main` funkcije

```
int main() {
    ifstream datoteka("tekst.txt");
    Tekst t(datoteka);
    while (true) {
        cout<<"Unesite trazenu rijec ili k za kraj:";
        string rijec;
        if (!(cin >> rijec) || rijec == "k")
            break;
        ispis(cout, t.upit(rijec)) << endl;
    }

    return 0;
}
```