Informatika 3
5. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.03.17.

Emlék: elemek törlése `list` tárolóban

- Egy `list`-ből a `remove` tagfüggvény segítségével kitörölhetjük azokat az elemeket, amik a paraméterként megadott értékkel egyenlők:

```
list<int> li;
...
li.remove(0); //törli a listából az összes 0-t
```

- Ha egy bizonyos feltételt teljesítő elemeket szeretnénk törölni a listából, akkor az egyik lehetőség, hogy egy iterátorral végigmegyünk rajta, és elemenként ellenőrizzük a feltételt, és ha teljesül, törlünk.
- Ez meglehetősen körülményes megoldás, hiszen az adott elemnek az `erase(iterator)` tagfüggvénnyel való törlése után az iterátor érvénytelen lesz, így nem tudjuk elérni a következő elemet.

Elemek törlése `list` tárolóban

- Egy lehetséges megoldás a következő:

```
list<int> li;
...
list<int>::iterator it = li.begin();
while (it != li.end()) {
    if (feltétel) {
        li.erase(it++);
    }
    else {
        ++it;
    }
}
```

- A törlésnél az `it++` a következő elemre lépteti az iterátort, de még az előző elemre mutató iterátort adja vissza, így a megfelelő elem törlődik, de elérjük a következő elemet is.

Elemek törlése `list` tárolóban

- Egy kényelmesebb megoldást kínál a `remove_if` tagfüggvény:

```
bool check(int) {...}
...
list<int> li;
...
li.remove_if(check)
```

- Itt a `remove_if` paramétere egy *függvénypointer*, amely egy olyan függvényre mutat, ami `bool` értéket ad vissza, és olyan típust vár paraméterként, amiket a `list`-ben tárolunk.
- A paraméterként megadott függvény lefut a lista összes elemére, és amelyekre igazzal tér vissza, azok az elemek törlődnek a listából.

Függvénypointerek

- Tekintsük a következő függvényt:

```
bool is_divisible(int n, int m) {
    return n%m == 0;
}
```

- A függvény neve (`is_divisible`) egy mutató arra a memóriacímre, ahol a függvény tárolódik.
- Ezt a mutatót el is tárolhatjuk egy változóban:

```
bool (*funcp)(int, int) = is_divisible;
```

Itt `funcp` egy bool visszatérési értékű, két darab `int` típusú paramétert váró függvényre mutató pointer.

- Általános szintaxis:

```
visszatérési típus (*név)(paraméterek típusai)
    = függvénypointer;
```

Függvénypointerek

- Függvénypointer lehet egy függvény paramétere is, ahogy ez pl. a `list` tároló `remove_if` függvénye esetén is így van.
- Egy másik példa, amikor egy problémára különböző megoldásaink vannak, és azokat tesztelni szeretnénk:

```
void test(bool (*funcp)(int), ...) {
...
}
```

A fenti `test` függvényt meghívhatjuk különböző függvényekre mutató pointerekkel, és mindegyikre végigfut ugyanaz a kód (le lehet mérni pl. a gyorsaságot, pontosságot stb.):

```
test(fun1, ...);
test(fun2, ...);
```

- Tekintsük a következő problémát: tegyük fel, hogy egy függvény egy `int`-eket tároló `list`-ből ki szeretné törölni azokat a számokat, amik modulo m egy meghatározott maradékosztályba esnek, de maga ez a maradékosztály is paraméter (értéke tehát előre nem ismert).
- Itt a `remove_if` tagfüggvénnyel nem tudjuk az előzőekben bemutatott módon megoldani a problémát, hiszen a paraméterként várt függvénynek csak egyetlen paramétere lehet, holott nemcsak az ellenőrzendő számot, hanem a maradékosztályt is át kellene adni.
- Erre (is) nyújt kényelmes megoldást az ún. lambda kifejezés (lambda expression, C++11-től), amellyel helyben definiálhatunk egy függvényt, amely a paraméterein kívül hozzáférhet az adott blokkban látható egyéb változókhoz is!

- Szintaxis:

  ```
  [capture clause] (paraméterek) {lambda törzs};
  ```

- Érdemes első közelítésben úgy gondolni erre, mint egy függvényre, aminek a megadott paraméterek a paraméterei, és a törzsben lévő utasításokat hajtja végre.
- A visszatérési értéket nem adjuk meg előre, az a törzsben esetlegesen szereplő `return` utasítás utáni kifejezés típusa lesz (ha ilyen nincs, akkor `void`).
- Ha a capture clause üres, akkor egy tényleg csak egy függvényt definiálunk, egy megfelelő pointernek értékül is adhatjuk azt, pl.:

  ```
  bool (*fp)(int) = [](int x) { return x==0; };
  ```

- A lambda expression erőssége, hogy azok a változók is átadhatók neki érték vagy referencia szerint, amik abban a blokkban láthatók, amelyben a lambdát definiáljuk:
  - `[=]` - minden külső változót érték szerint rögzítünk;
  - `[&]` - minden külső változót referencia szerint rögzítünk;
  - `[x, &y]` - az `x` változót érték szerint, az `y`-t referencia szerint rögzítjük, más külső változó nem látható;
  - `[&, x]` - minden külső változót referencia szerint rögzítünk, kivéve az `x`-et, amit érték szerint;
  - `[=, &x]` - minden külső változót érték szerint rögzítünk, kivéve az `x`-et, amit referencia szerint;
  - `[this]` - osztály tagfüggvényében átadhatjuk a `this` mutatót, hogy hozzáférjünk az adott példány tagváltozóihoz;
  - `[*this]` - osztály tagfüggvényében átadhatjuk a `this` mutatót érték szerint (C++17-től).

- Ha egy lambda expression-t hozunk létre, és a capture clause nem üres, akkor minden esetben egy egyedi típus jön létre, és ha ezt egy változóban el akarjuk tárolni, akkor az auto kulcsszót kell használnunk:

```
int x = 5;
auto lambda = [x](int y) { return x + y; };
cout << lambda(4) << endl; // az eredmény "9"
```

- Az auto kulcsszó máshol is használható a típus megjelölésére, ekkor az értékadás alapján a fordító automatikusan detektálja a típust, pl.:

```
vector<list<unsigned int>> v;
...
for(auto li : v) {...}
```

- Az auto kulcsszó értékadás nélkül nem használható:

```
auto x; //HIBA
```

- Ha mi magunk szeretnénk megadni a visszatérési típust (felül szeretnénk bírálni a fordítót), akkor megadhatunk egy záró visszatérési típust:

```
int x = 5;
auto lambda = [&]() { return ++x; };
++lambda(); //HIBA: a fordító a ++x által visszaadott
            //referenciát int-té konvertálja
```

A működő verzió:

```
int x = 5;
auto lambda = [&]() -> int& { return ++x; };
++lambda(); //MŰKÖDIK: a lambda visszatérési értéke
            //egy x-re mutató referencia
```

- Az érték szerint rögzített külső változók a lambdában nem változtathatók meg, a következő kód nem fordul:

```
int x = 5;
auto lambda = [x]() { return ++x; };
```

- Úgy kell elképzelni, mintha létrejönne egy osztály egy konstans `x` változóval, aminek az értéke 5.
- A `mutable` kulcsszóval ezek a változók módosíthatók lesznek:

```
int x = 5;
auto lambda = [x]() mutable { std::cout << ++x; };
lambda();
x = 10;
lambda();
```

Mit ír ki a fenti kód?

- A kezdeti problémánk megoldása lambdával:

```
void delete_if_k_mod_m(list<int>& li, int k, int m) {
    li.remove_if([k, m](int i) {
                    return (i - k) % m == 0;
                });
}
```

A fenti függvény kitörli a `li` lista elemei közül azokat amik `k`-val kongruensek modulo `m`.

for_each

- A függvénypointerek ill. a lambda expression kényelmesen alkalmazhatók akkor is, ha végre szeretnénk hajtani valamilyen műveletet egy tároló összes (vagy több) elemén.
- Ekkor az `algorithm` fejállományban található `for_each` függvény használható.
- Szintaxis:

```
for_each(kezdő iterátor, záró iterátor,
                              függvénypointer/lambda);
```

- Az alkalmazott függvény paramétere olyan típusú kell legyen, mint amit a tárolóban tárolunk.

A `map` tároló

- A `vector` tároló tekinthető egy függvénynek a $\{0, 1, \ldots, n-1\}$ halmazon, ahol n a `vector` elemeinek száma.
- A `map` fejállományban található `map` tároló egy ilyen irányú általánosítása a `vector`-nak, ahol az értelmezési tartománynak nem kell egymást követő számokból állnia, sőt, az tetszőleges típusú objektumokból állhat.
- A deklaráció a következőképp néz ki:

  ```
  map<key_type, value_type> m;
  ```

  Itt a `key_type` tetszőleges objektum lehet, amelyre definiálva van a < operátor.
- Egy `map` tehát *kulcsokhoz* rendel *értékeket*, vagyis a kulcsok halmazán értelmezett függvénynek tekinthető.

A map tároló

- A `k` kulcshoz a `v` érték a következőképp rendelhető:

  ```
  m[k] = v;
  ```

- Ha már definiálva van a `k` kulcsra az érték, akkor a fenti sor felülírja azt.
- Ha egy kulcshoz még nincs érték definiálva, akkor a következőképp is megtehetjük ezt:

  ```
  m.insert(make_pair(k,v));
  ```

  A paraméter egy `pair<key_type, value_type>` objektum, ha már definiálva van az érték, akkor ez a kód nem csinál semmit.
- Definiál vagy felülír a következő függvény is:

  ```
  m.insert_or_assign(k,v);
  ```

A `map` tároló

További tagfüggvények:

- Az `m.count(key_type)` függvény 1-et ad vissza, ha a paraméterként megadott kulcsra már definiált az érték, és 0-t különben.
- Az `m[i]` kód visszaadja az `i` kulcshoz tartozó értéket, ha az definiált, és valamilyen default értéket definiál különben (beépített típusok esetén 0-t, egyébként lefut a default konstruktor).
- Mi történik, ha nincs publikus default konstruktor?

  Nem fordul le a program még akkor sem, ha amúgy az érték definiált.
- Helyette használhatjuk az `at(key_value)` tagfüggvényt, ez ún. kivételt dob, ha nincs definiált érték (erről később tanulunk).

A map tároló

További tagfüggvények:

- Az `m.erase(key_type)` függvény törli a megadott kulcshoz tartozó értéket, ha definiált, különben nem történik semmi.
- Az `size()`, `clear()`, `empty()` tagfüggvények az korábbiaknak megfelelően működnek.
- A `map` a szokásos módon iterálható, az egyes elemek `pair` objektumokként tárolódnak:

```
for(pair<key_type, value_type> p : m){
    key_type key = p.first;
    value_type value = p.second;
}
```

A map tároló

Egy példa: definiáljuk az a_n ($n \in \mathbb{N}^+$) sorozatot a következő rekurzióval: legyen $a_1 = 1$, és $n > 1$ esetén

$$a_n = \sum_{d|n,\, d<n} a_d.$$

Írjunk függvényt, ami visszaadja a_n értékét egy megadott n-re.

- Ha már egyszer kiszámoltunk egy értéket, azt nem kell mindig újraszámolni, ha eltároljuk.
- Általában egy a_n meghatározásához nem kell ismernünk az összes a_k értéket n-nél kisebb k-ra, pusztán n osztóira, tehát általában felesleges helyet n helyet foglalni az értékeknek
  $\rightsquigarrow$ Jól használható a `map` tároló.

A static kulcsszó

- Ahhoz, hogy eltároljuk a már kiszámolt értékeket, nem kell a függvényt és a map-et egy objektumba csomagolni.
- Ha egy függvényben a static kulcsszót használjuk a függvény lokális változójának deklarációjánál, akkor az a függvénytörzs futása után nem törlődik, a tartalma a függvény következő futásainál is elérhető:

```
int recur(int n) {
    static map<int,int> lookup;
    ...
}
```

- Egy hátránya a fenti megoldásnak, hogy a lookup tartalma nem törlődik akkor sem, ha már nincs rá szükségünk.
- Egy lehetséges megoldás a fenti problémára, ha negatív paraméterrel hívva a függvényt töröljük az adatokat:

```
if(n < 0) lookup.clear();
```

A `static` kulcsszó

- A `static` kulcsszó használható függvények deklarációjánál is.
- (Itt most nem osztályok tagfüggvényeiről van szó, azt az esetet később tárgyaljuk.)
- Ha egy függvényt `static`-ként deklarálunk, akkor más file-ból nem lesz elérhető.
- Megfordítva: ha a függvényt nem akarjuk más file-ból elérni, akkor deklaráljuk `static`-ként!

A static kulcsszó

A következő függvény két int szorzatát számolja ki modulo m:

```
static int mult_mod_m(int a, int b, int m) {
    return (a * b) % m;
}

int main(){
...
    int a = 5;
    int m = 7;
    int b = mult_mod_m(a, a, m);
...
}
```

A `static` kulcsszó

- A fenti kód jól olvasható, de ha szó szerint ez fut le, akkor felesleges lépéseket végzünk: lemásoljuk a paraméterül adott változókat a lokális változókba (`a`-t ráadásul kétszer), és a függvényhívásnak/visszatérésnek is van költsége.
- Szerencsére a fordító okos, és ha segítünk neki, akkor tud optimalizálni.
- Mivel `static`-ként deklaráltuk a függvényt, a fordító tudja, hogy más file-ból nem fogjuk azt meghívni, nem történik tehát semmi baj, ha a lefordított programban nincs ilyen függvény, hanem az egyszerű függvénytörzset a függvényhívás helyére másolja, ezzel gyorsítva a program futását.
- A fordító dönti el, hogy érdemes-e ezt megcsinálni (tipikusan rövid függvényeknél érdemes), de a programozónak meg kell adnia azt a segítséget, hogy `static`-ként deklarálja a függvényt (másképp nem lehet optimalizálni).

`inline` függvények

- Ha egy függvény egy külön fejállományban van deklarálva, és egy attól különböző cpp-ben van definiálva, akkor a fordító nem tudja elvégezni ezt az optimalizálást.
- A cpp file külön fordítási egység, és a fordítás első szakaszában a függvényhívásnál a fordító csak azt látja, hogy milyen függvényt használ majd a program, de annak definícióját nem.
- Ha viszont a fejállományban definiáljuk a függvényt, és azt több cpp file is használja, akkor a fordítás során ugyanannak a függvénynek több definíciója is szerepelni fog.
- Ez fordítási hibát eredményez, hiába egyeznek meg a definíciók (ezt a fordító nem vizsgálja), mert egy függvénynek csak egy definíciója lehet (one definition rule - ODR).

`inline` függvények

- A C++-ban ezt a problémát ún. `inline` függvények segítségével kezelhetjük, ehhez az `inline` kulcsszót írjuk a függvény elé:

```
inline void f(){
...
}
```

- Ilyen esetben a fordító tudja, hogy egy esetleges többszörös definíció esetén az ok a függvény inline volta, és így a definíciók megegyeznek.
- Ekkor van *lehetőség* optimalizálásra, de arról, hogy pontosan hogyan fordul a kód, a fordító (és nem a programozó) dönt.

`inline` függvények

- Tipikusan nagyon rövid és sokszor meghívott függvényeknél történik optimalizálás, amikor egy függvényhívás ideje összehasonlítható a függvénytörzs futásának idejével, ilyenkor az `inline` függvények használata gyorsabb futásidőt eredményezhet.
- Ilyenek példák az osztályok rövid konstruktorai vagy egyszerű tagfüggvényei.

inline függvények

- Az osztály definíciójában definiált függvények automatikusan inline függvények, itt a kulcsszó használata opcionális.

```
class C{
    C() {...} //implicit inline konstruktor
};
```

- Ha az osztály definícióján kívül adjuk meg a függvénydefiníciót, az inline kulcsszót elég a függvénydefiníciónál használni.

```
class C(){
    ...
    void f();
};

inline void C::f() {
...
};
```

`inline` függvények

- Az `inline` függvényeket a fejállományban kell definiálni (kivéve, ha csak egyetlen cpp file-ban használjuk őket).
- Az `inline` függvények használatának egy hátránya, hogy nagyobb file-okat generálhat a fordító, ill. a fordítási idő nőhet.
- Vannak esetek, amikor kötelező az `inline` függvények használata, ezekről később lesz szó.

a friend kulcsszó

- Egy osztály `private` tagváltozói és tagfüggvényei csak az osztály számára elérhetők.
- Néha azonban hasznos lehet, ha más függvények is hozzáférhetnek ezekhez.
- A `bme::Point` osztály mellett definiáltuk a `dist` és `midpoint` függvényeket, ezek a `getX()` és `getY()` tagfüggvényeken keresztül fértek hozzá a `Point` osztály változóihoz.
- Az osztályban a `friend` kulcsszó segítségével megadhatunk olyan függvényeket, amik hozzáférhetnek a `private` tagokhoz:

```
class Point {
...
    friend double dist(Point P, Point Q);
};
```

- Ezután a `dist` függvény hozzáfér a `private` tagokhoz.
- Nem csak függvény, hanem osztály is lehet `friend`!

a `friend` kulcsszó

- **Fontos: a fenti kód nem függvénydeklaráció!** A függvényt ezután az osztályon kívül deklarálni kell:

```
class Point {
...
    friend double dist(Point P, Point Q);
};
...
double dist(Point P, Point Q);
```

- A `friend double dist(Point P, Point Q);` sor azt jelenti, hogy ha majd lesz egy ilyen `dist` függvény, akkor az hozzáférhet a `private` tagokhoz.
- Sajnos a legtöbb esetben a program lefordul deklaráció nélkül is, ezért könnyű elfeledkezni róla. Ennek okáról pl. itt olvashatunk.

a `friend` kulcsszó

- A `friend` függvények használatának egyik haszna, hogy a tagváltozók közvetlen elérése gyorsabbá teszi a programot.
- Ez akkor eredményez számottevő gyorsulást, ha az adott függvényt sokszor hívjuk meg.
- Megjegyzés: szintén gyorsíthatunk a programon, ha referenciákat adunk át paraméterként a függvénynek, ekkor ugyanis a változók másolása nem történik meg.