Informatika 3
8. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.04.28.

Függvényobjektumok

- Sok algoritmus a vektorokat, listákat stb. csak iterátorok és értékek segítségével kezeli.
- A következő kód az `algorithm` fejállományban található `find` függvény segítségével keresi meg a `li` listában az első 7-es értéket:

```
void f(list<int>& li){
    list<int>::iterator it =
                find(li.begin(), li.end(), 7);
    ...
}
```

Függvényobjektumok

- Korábban a `list` objektum `remove_if` tagfüggvénye esetén láttunk példát arra, amikor mi adunk meg egy függvényt, amit az algoritmus használ.
- A `find_if` függvény segítségével (annak egy függvénypointert megadva) az előző kódot úgy módosíthatjuk, hogy az az első 7-nél kisebb elemet találja meg:

```
bool less_than_7(int i) {
    return i < 7;
}

void f(list<int>& li){
    list<int>::iterator it =
        find_if(li.begin(), li.end(), less_than_7);
    ...
}
```

Függvényobjektumok

- Előfordul, hogy a meghívott függvényt paraméterezni kell, vagy annak el kell tárolnia a futások közt valamilyen információt.
- Pl. elvárhatjuk, hogy a fenti példában a 7 korlátot a felhasználó állíthassa be, ill. hogy ne csak az első, a feltételeknek megfelelő elemet keresse meg az algoritmus, hanem gyűjtse össze egy tárolóba az összes ilyen elemet.
- A `find_if`, `remove_if`, `for_each` stb. függvényeknek nem csak függvénypointereket adhatunk paraméterül, hanem tetszőleges olyan kifejezést, ami meghívható a megfelelő paraméterrel.
- Ilyen példa volt korábban a lambda expression.

Függvényobjektumok

- A *függvényobjektumok* (vagy *funktorok*) olyan objektumok, amikkel a fenti célok (paraméterezhetőség, futások között az adatok tárolása) kényelmesen megvalósíthatók.
- Magát az alkalmazandó függvényt az objektum `operator()` tagfüggvényében írjuk meg, az objektum tagváltozóiban pedig tárolhatók a paraméterek ill. eredmények.

Függvényobjektumok

- Pl. a következő objektum konstruktorában megadhatunk egy korlátot, amelynél kisebb elemeket gyűjt:

```
template<typename T>
class Collector {
private:
    T bound;
    list<T> res;
public:
    Collector(T _bound) : bound(_bound) {}
    void operator()(T el) {
        if (el < bound) {
            res.push_back(el);
        }
    }
    const list<T>& result()const { return res; }
};
```

Függvényobjektumok

- A `for_each` függvénynek megadhatunk egy ilyen objektumot (akár temporálisat is, mint az alábbi példában), ezt aztán visszatérési értékként megkapjuk:

```
list<int> li = { 1,2,3,4,5,6 };
Collector<int> c =
    for_each(li.begin(), li.end(), Collector<int>(4));
list<int> res = c.result();
```

- A fenti kód kigyűjti a 4-nél kisebb számokat a listából, majd azokat átmásoljuk a `res` listába.

Függvényobjektumok

- Korábban is találkoztunk már ilyen függvényobjektummal a sztenderd osztályból, amikor egy tömböt rendeztünk:

```
int tomb[] = { 1, 2, 10, -4, 6 };
sort(tomb, tomb + 5, greater<>());
```

- Itt a `greater<>()` objektum valósítja meg az összehasonlítást.
- Az üres template azért van, mert itt a `greater<void>` specializációt hívjuk meg, amelynek az `operator()` függvénye template, és az azonosítja be a típust.
- Írhatnánk a következőt is:

```
int tomb[] = { 1, 2, 10, -4, 6 };
sort(tomb, tomb + 5, greater<int>());
```

Függvényobjektumok

- Tegyük fel, hogy az `Alakzat` objektum rendelkezik egy virtuális rajzolófüggvénnyel:

```
class Alakzat {
...
public:
    virtual void rajzol(int szin) = 0;
...
};
```

- Szeretnénk egy `Alakzat` objektumokra mutató pointereket tároló `vector` összes elemére meghívni a `rajzol` függvényt a `for_each` függvény segítségével.

Függvényobjektumok

- A függvényhívó objektum:

```
template<typename R, typename S, typename T>
class mem_fun_call {
    R(S::* pmf)(T);
    T t;
public:
    mem_fun_call(R(S::*p)(T), T _t) : pmf(p), t(_t) {}
    R operator()(S* p)const { return (p->*pmf)(t); }
};
```

- A `pmf` tagváltozó egy, az `S` osztály tagfüggvényére mutató függvénypointer.
- A `t` változó tárolja a függvénynek átadott paraméter értékét.
- Ezeket a konstruktorban adjuk meg az osztálynak.
- Az `operator()` megkap egy példányra mutató pointert, aminek a megfelelő tagfüggvényét meghívjuk a megfelelő paraméterrel.

Függvényobjektumok

- Használata:

```
vector<Alakzat*> v;
...
for_each(v.begin(), v.end(),
      mem_fun_call<void, Alakzat, int>
                (&Alakzat::draw,0x0000FF));
```

- A template paramétereket a fordító is kikövetkezteti:

```
for_each(v.begin(), v.end(),
    mem_fun_call(&Alakzat::draw,0x0000FF));
```

Az enum típus

- A programokban gyakran használunk olyan egész típusú konstans értékeket, amelyek logikailag összetartoznak.
- A kód olvashatóságát növeli, ha ezeket az értékeket nevekkel helyettesítjük.
- Az egyik megoldás konstansok használata.
- A másik megoldás egy új típus, egy ún. enumeration (felsorolás) definiálása az értékkészletének megadásával:

```
enum Level {
  LOW, MEDIUM, HIGH
};
```

- Az `enum` kulcsszó után kerül a típus neve (a típusazonosító), ennek elhagyásával típus nem jön létre, csak a konstansok születnek meg.

Az enum típus

- A felsorolásban szereplő nevek mindegyike egy-egy egész számot jelöl.
- Alapértelmezés szerint az első elem (`LOW`) értéke 0, a rákövetkező elemé (`MEDIUM`) 1 stb.
- A felsorolásban az elemekhez közvetlenül értéket is rendelhetünk:

```
enum Level {
  LOW=2, MEDIUM=10, HIGH
};
```

- A fenti felsorolásban a `LOW` értéke 2, a `MEDIUM` értéke 10, a `HIGH` értéke 11 lesz.
- Egy `enum` típus létrehozása után deklarálhatunk egy ilyen típusú változót:

```
enum Level myVar;
```

Az enum típus

- Bár az `enum` egész típus, egy `Level` típusú változó értéke csak valamelyik, a felsorolásban szereplő konstans lehet a felsorolásban szereplő névvel megadva.

```
myVar = 2;          //Hiba
myVar = MEDIUM; //OK
myVar = Level::MEDIUM; //OK
```

- Kiírásnál a változó egész értéke (az alábbi esetben 10) jelenik meg.

```
myVar = MEDIUM;
cout << myVar << endl;
```

- `switch` utasításban alkalmazható:

```
switch(myVar){
 case LOW: ...
 case 10: ...
 ...
```

Bitenkénti műveletek

- Az `enum` típust gyakran használják paraméterezésre a bitenkénti logikai műveletek segítségével.
- Ha egy változó értékét pl. egy byte-on tároljuk, akkor az leírható egy 8 hosszú bitsorozattal.
- A `|` (bitenkénti vagy), `&` (bitenkénti és), `^` (bitenkénti kizáró vagy), `~` (bitenkénti nem) a logikai műveleteket az operandusokon bitenként végzik.
- Pl. ha egy `i int` típusú változó értéke 5, azt a

  00000000 00000000 00000000 00000101

  bitsorozat reprezentálja (kettes számrendszerben ábrázolja).
- Így `i|3` értéke 7 (a 3 integer literál), `i&6` értéke 4, az `i^3` értéke 6 stb.
- Léteznek a `|=`, `&=`, `^=` operátorok is.

Bitenkénti műveletek

- Ennek segítségével egy `enum` típus elemei (vagy akár egész konstansok) kapcsolóként használhatók.
- Pl. ha különböző típusú gráfokat egy osztállyal írunk le, akkor egyes bitekhez rendelhetünk tulajdonságokat: az első bit leírhatja, hogy tárolunk-e éllistát, a második, hogy tárolunk-e szomszédsági mátrixot, a harmadik, hogy egyszerű-e a gráf, a negyedik, hogy irányított-e, az ötödik, hogy súlyozott-e stb.
- Ezekhez definiáljuk a következő `enum`-okat

```
enum GraphProperties {
    EDGE_LIST = 1,
    ADJACENCY_MATRIX = 2,
    SIMPLE = 4,
    DIRECTED = 8,
    WEIGHTED = 16
    ...
};
```

Bitenkénti műveletek

- Ezek után a `Graph` osztály tartalmazhat egy `properties` `int`-et, egy konstruktorban pedig beállíthatjuk ezt:

```
class Graph{
    int n;
    int properties;
...
    Graph(int _n, int _prop = (EDGE_LIST | SIMPLE)) :
            n(_n), properties(_prop){}
```

- A fenti konstruktorban a properties alapértelmezett értéke 5, aminek az ábrázolásában az 1. és a 3. bit 1-es.
- Ezek után az éllistás tárolást a következőképp kérdezhetjük le: `if(properties & EDGE_LIST) {...}`
- Ha nem állítottak be sem éllistát, sem szomszédsági mátrixot, akkor beállíthajuk az éllistát: `properties |= EDGE_LIST;`

Bitenkénti műveletek

- További bitenkénti műveletek a jobb shift (>>) és bal shift (<<), ezek eltolják a bitsorozatot jobbra ill. balra a megadott számú hellyel (a bejövő új bitek mindig 0-k).
- ```
  int i = 3;
  cout << (i<<3); // 3 hellyel balra tolunk
                  // = szorzás 8-cal
                  // az eredmény 24
  ```
- ```
  int i = 5;
  cout << (i>>2); // 2 hellyel jobbra tolunk
                  // = osztás 4-gyel
                  // az eredmény 1
  ```
- A shiftek segítségével nagyon gyorsan lehet 2 hatványaival szorozni és osztani.

Véletlenszám-generálás

- A C++-ban a `cstdlib` fejállományban lévő `rand()` függvény segítségével generálhatunk véletlen értékeket.
- A `rand()` függvény egy nemnegatív számot ad vissza, a maximális lehetséges érték a `RAND_MAX` kifejezés értéke.
- A `RAND_MAX` értéke lehet pl. $2^{15} - 1$ (különböző megvalósításoknál lehet különböző).
- Fontos: a `rand()` függvény nem egy véletlenszerű számot generál, hanem egy úgynevezett *pszeudorandom* számot, ami *úgy viselkedik*, mintha egy egyenletesen véletlenszerűen választott egész szám volna a [0; `RAND_MAX`] intervallumon.
- Egy inicializálás után a `rand()` függvény viselkedése determinisztikus, azaz ugyanúgy inicializálva mindig ugyanazokat a számokat generálja egymás után.
- Előny: reprodukálható az algoritmus futása.

Véletlenszám-generálás

- Vigyázni kell arra, hogy a `rand()` függvény által generált számok bizonyos függvényei nem feltétlenül viselkednek véletlenszerűen (úgy, ahogy várnánk).
- Pl. a számok azon bitje, ami a mod 2 maradékokat reprezentálja, nem feltétlenül viselkedik úgy, mintha egy szabályos érmedobás eredményeit kódolná.
- A $(0, 1)$ intervallumon egyenletes eloszlás generálása: osszuk le a `rand()` által generált értéket a lehetséges értékek intervallumának hosszával (a határokat is kezeljük).
- Az (a, b) intervallumon egyenletes eloszlás generálása: skálázzuk az előző pontban kapott értéket.
- Egy szám választása az $\{1, \ldots, n\}$ halmazban egyenletesen véletlenszerűen: vegyük a $(0, 1)$-en egyenletesen választott szám n-szeresének egészrészét, adjunk hozzá 1-et.

Véletlenszám-generálás

- Az `srand()` függvény segítségével inicializálhatjuk véletlenszám-generátort.
- Egy `unsigned int` paraméterrel megadhatjuk a mag értékét, amiből kiindulva a `rand()` az értékeket generálja.
- Az `srand(érték)` utasítás után a `rand()` futása determinisztikus.
- A mag értékét sokszor választják az aktuális időpont alapján: pl. a `ctime` fejállományban található `time(0)` utasítás visszaadja egy bizonyos fix időpont óta eltelt másodpercek számát (visszatérési érték típusa `time_t`, `unsigned int`-té konvertálható).
- Erre különböző másodpercekben beállítva a magot más-más futást kapunk (ugyanazon másodpercben beállítva ugyanolyan futást kapunk).