Informatika 3
6. előadás

Tóth Dávid

Budapesti Műszaki és Gazdaságtudományi Egyetem

2026.03.24.

`static` osztályváltozók és -függvények

- Ha egy változó magához az osztályhoz, és nem az egyes példányokhoz tartozó értéket tárol, akkor `static`-ként kell azt deklarálnunk:

```
class C{
    static int stat_val;
};
```

- Ennek értéke (ha a változó publikus) a következőképp érhető el:

```
C::stat_val;
```

- Az osztály példányain keresztül is elérhetők a statikus változók:

```
C variable;
variable.stat_val;
```

`static` osztályváltozók és -függvények

- Hasonlóan kezelhetők az osztályhoz (és nem a példányokhoz) tartozó függvények:

```
class C{
    static int stat_func();
};
```

- Elérés:

```
C::stat_func();

C variable;
variable.stat_func();
```

- `static` változók elérhetők `static` függvényekből, de egyéb (a példányokhoz tartozó) változók nem!

Parancssori argumentumok

- Egy program futtatásának kezdetén megadhatunk ún. parancssori argumentumokat is.
- Ezek a `main` függvény következő deklarációjában a függvény argumentumaiként érhetők el:

```
int main(int argc, char** argv){
...
}
```

- A `main` első argumentuma a parancssori argumentumok száma.
- Ez mindig legalább 1, az első parancssori argumentum a program neve.
- A második argumentum tömbökre mutató pointerek tömbje, `argv[i]` az `(i+1)`-dik argumentumot tartalmazó tömb elejére mutat.

A switch utasítás

- Ha egy kifejezés értéke szerint szeretnénk elágazást írni, akkor használhatjuk a switch utasítást.
- A szintaxis:

```
switch(kifejezés) {
  case x:
    // kód blokk
    break;
  case y:
    // kód blokk
    break;
  ...
  default:
    // kód blokk
}
```

A `switch` utasítás

- A `switch` utasítás után zárójelben áll az a kifejezés, amelynek az értéke szerinti elágazást írjuk.
- `{` és `}` közt írjuk az egyes értékekhez tartozó részt.
- A `case x:` sor után következő sorok fognak végrehajtódni, ha a kifejezés értéke `x`, stb.
- A `break;` utasításhoz érve a program kilép a `switch`-ből.
- Fontos: ha nincs `break`, a program fut tovább az esetleg más esetekhez tartozó sorokkal (ezt persze előnyünkre is fordíthatjuk).
- Lehetőség van arra, hogy a felsoroltaktól eltérő értékek esetét is kezeljük. Ilyen esetben a `default:` után következő sorok futnak le. Ha ilyen kulcsszó nem szerepel, akkor nem történik semmi ezekben az esetekben.

A `break` és `continue` utasítások

- A `break` utasítással nem csak a `switch` után következő blokk futását szakíthatjuk meg.
- Ugyanilyen funkcióval használhatjuk a `for`, `while` vagy `do-while` ciklusok esetén.
- A `continue` utasítás a fenti három ciklus esetén használható: megszakítja a ciklusmag futását, és amennyiben a ciklusfeltétel teljesül, újraindul a futás a ciklusmag elejéről (`for` esetén a növelő utasítás is végrehajtódik).

A objektumok életciklusa

- Egy objektum életciklusa a létrehozásával kezdődik, amikor memória foglalódik az objektum számára.
- Ezután meghívódik a konstruktor, ami inicializálja az objektumot.
- A tagváltozók konstruktora abban a sorrendben fut le, amilyen sorrendben megadtuk őket az osztályban.
- Amikor az objektum megszűnik, akkor lefut az ún. *destruktora*.
- Ha külön nem definiálunk destruktort, akkor üres törzzsel létrehoz egyet a fordító.
- A destruktor törzsének futása után az osztály tagváltozók destruktora fut le, fordított sorrendben, mint ahogy az osztályban megadtuk őket.

A objektumok életciklusa

- A destruktor működését az `IntegerArray` osztályon illusztráljuk.
- Az `IntegerArray` adattagjai

```
class IntegerArray {
private:
    int n;
    int* data;
...
```

- Az `IntegerArray` konstruktora a `new` kulcsszó segítségével foglal memóriát a `data` tömbnek.
- Ezt a memóriát nekünk kell felszabadítani, amikor az objektumot nem használjuk többet, azaz a destruktorban.

A objektumok életciklusa

- A destruktor deklarálása:

```
~IntegerArray();
```

- Definíciója a cpp file-ban:

```
IntegerArray::~IntegerArray() {
    delete[] data;
}
```

- Természetesen a destruktor is lehet `inline` függvény.

Az = operátor

- Ha `a` és `b` ugyanzaon osztály két példánya, akkor az `a = b;` művelet akkor is értelmes, ha az egy = operátort nem terheljük túl.
- Ilyen esetben a fordító egyszerűen átmásolja a `b` tagváltozóinak értékét az `a` tagváltozóiba.
- Ha nem ez az elvárt működés, akkor az = operátort túl kell terhelni.
- Példa: az `IntegerArray` osztályban az = operátor default működése alapján az `a.data` pointer ugyanarra a címre mutatna, mint a `b.data` pointer, tehát az `a` objektum megváltoztatása megváltoztatná `b`-t is.
- Az `a = b;` utasítás elvárt működése, hogy az `a` példány ugyanazt reprezentálja, mint a `b` példány, de amúgy a két példánynak egymással további kapcsolata ne legyen.

Az = operátor

- Az = operátor paramétere egy `const` referencia egy azonos típusú osztályra.
- Az `a = a;` parancs értelmes, és itt az a tagfüggvénye hívódik meg, ezért hiába konstans a paraméter, az a értéke változhat.
- Ez hibás működést eredményezhet, ezt az esetet kezelni kell:

```
IntegerArray& IntegerArray::operator=(const
                                      IntegerArray& other) {
    if (this != &other) {
        delete[] data;
        n = other.n;
        data = new int[n];
        for (int k = 0; k < n; k++) {
            data[k] = other.data[k];
        }
    }
    return *this;
}
```

A másoló konstruktor

- Ha mi nem tesszük meg, akkor minden osztályhoz definiál a fordító egy másoló konstruktort, aminek a paramétere egy azonos típusú osztályra mutató konstans referencia, és a paraméter tagváltozóinak értékeit átmásolja a saját tagváltozókba.
- Ez hívódik meg pl. akkor, ha egy függvénynek nem referenciákat adunk meg paraméterként.

```
bme::Point dist(bme::Point P, bme::Point Q){...}
```

  A paraméterül megadott változók átmásolódnak a lokális változókba.
- Az alábbi második deklaráció is másoló konstruktort hív:

```
bme::Point P(-1.0,2.0);
bme::Point Q(P);
```

A másoló konstruktor

- Ha a fentitől különböző működésre van szükség, akkor felüldefiniálhatjuk a másoló konstruktort:

```
IntegerArray::IntegerArray(const IntegerArray& other){
    n = other.n;
    data = new int[n];
    for (int k = 0; k < n; k++) {
        data[k] = other.data[k];
    }
}
```

Template-ek

- Tegyük fel, hogy akarunk írni egy függvényt, ami három paraméterül megadott szám közül visszaadja a legnagyobb értékét. Pl.

```
int max_of_three(int a, int b, int c) {
  ...
}
```

- Probléma: különböző típusok esetén külön függvényre van szükségünk, ráadásul ha később változtatni szeretnénk az algoritmuson, akkor minden verziót meg kellene változtatni.
- Erre a problémára kínálnak megoldást a C++-ban a *sablonok* (template-ek), amik alapján a fordító *generálja* a függvény különböző overload-jait.

Template-ek

- A fenti függvényhez pl. a következőképp lehet sablont készíteni:

```
template<typename T>
T max_of_three(T a, T b, T c) {
  ...
}
```

- Szintaxis:
  - a `template` kulcsszó jelzi, hogy sablon következik,
  - a `template` kulcsszó után kacsacsőrök között adjuk meg a template paramétereit,
  - template paraméterként a függvénynek típusokat (`typename`) adunk, ezeket fogja a fordító igény szerint `int`-tel, `double`-lel stb. helyettesíteni,
  - a függvényben a `T`-t már használhatjuk típusként,
  - `template<typename T>` helyett `template<class T>` is írható, a két megoldás ekvivalens.

Template függvények hívása

- Template függvények hívásánál a függvénynév után kacsacsőrök között adjuk meg, hogy milyen típussal hívjuk meg a függvényt:

```
int main(){
    int i = max_of_three<int>(2,3,4);
    dobule d = max_of_three<double>(2.,3.,4.);
}
```

- Ha a template paramétere szerepel függvényparaméter típusaként, akkor a fordító le is tudja vezetni a típust a megadott paraméterek típusából (amennyiben azok pontosan passzolnak), így azt nem szükséges megadni:

```
int main(){
    double d = max_of_three(2.,3.,4.);
    //OK, a fordító levezeti a típust
}
```

Template függvények hívása

- A template függvények olyan típusokkal működnek, amiket behelyettesítve értelmes kódot kapunk.
- Ha pl. < operátort használunk a függvényben, akkor olyan típusnál, aminek nincs ilyen operátor overload-ja, fordítási hiba keletkezik.
- Mit ír ki a következő kód?

```
template<typename T>
T duplaz(T const& x) {
    return x+x;
}

int main() {
    std::cout << duplaz(3) << std::endl;
    std::cout << duplaz(std::string("ha"))
              << std::endl;
}
```

Template osztályok

- Típusokat osztályokban is paraméterezhetünk, azaz lehet template osztályokat definiálni.
- Tipikusan ilyenek a sztenderd könyvtár tárolói: a `vector`, a `list`, a `set` stb.
- Itt is a felhasználó adja meg, hogy milyen típusú objektumokat tárolunk, a fordító pedig minden egyes típushoz legyárt egy külön tárolóosztályt.
- Ilyen a `complex` osztály is, ahol megadhatjuk, hogy a valós és képzetes részeket milyen típusú elemekkel írjuk le: `complex<double>`, `complex<int>` stb.

Komplex számok

- A `complex` fejállományban,
- a deklarálásnál meg kell adnunk <> között, hogy milyen típusúak a valós és képzetes részek, pl.
  `complex<double> z;`
  `complex<double> z(.5,.5); //0.5+i*0.5 kezdeti értékkel`
  `z=complex<double>(1.5,.5);`
- valós rész: `z.real()`,
- képzetes rész: `z.imag()`,
- ha fárasztó a sok gépelés, megadhatunk egy rövidítést a következőképp:
  `typedef complex<double> C`,
  innentől `C` egy `double` valós- és képzetes részű komplex számot jelöl a programban,
- a szokásos műveletek definiáltak.

Template osztályok

- A template osztályok létrehozásánál a szintaxis hasonló, mint a függvénynél:

```
template<typename T, typename U>
class MyPair {
private:
    T a;
    U b;
 ...
};
```

- A template paraméterekből több is lehet, ezek a paraméterek lehetnek értékek is (non-type), beépített típusok, egész típusok (`float, double` nem), de fontos megkötés, hogy az értéküknek fordítási időben ismertnek kell lenni.

Template osztályok tagfüggvényei

- A template osztályok tagfüggvényeit is definiálhatjuk az osztályon kívül is.
- Ilyen esetben ezt az osztályt tartalmazó fejállományban kell megtenni, és a template tagfüggvények mindig `inline` függvénynek minősülnek (az `inline` kulcsszó szerepeltetése nem kötelező).
- Szintaxis:

```
template<typename T, typename U>
void my_pair<T,U>::SetVal(const T& a, const U& b) {
    ...
}
```

- A template osztályokat tartalmazó file-ok konvenció szerint `hpp` kiterjesztést kapnak.

Template osztályok tagfüggvényei

- Nem lehet külön cpp file-ban definiálni: ezek ugyanis külön fordítási egységek, és a fordító számára nem lenne világos, hogy milyen template paraméterhez tartozó osztály függvényének a megvalósításával van dolga.
- Másképp: a cpp fordításakor a fordítónak fogalma sincs arról, hogy mi a `main`-ben milyen template paraméterrel hívtuk meg az adott függvényt.
- Amikor egy konkrét template paraméterrel deklarálunk egy osztálypéldányt, akkor a fordítónak hozzá kell férnie a tagfüggvények definíciójához a fejállományban.

Template osztályok tagfüggvényei

- Egy függvénynek egy programban egyetlen definíciója lehet, így a linker (ami egy futtatható programmá egyesíti a fordító által készített objektumfájlokat), tudja, hogy egy függvényhíváshoz milyen definíció tartozik.
- De: a template objektum tagfüggvényei inline függvények kell legyenek: így ugyan esetleg többször lefordul a függvény, de erről a linker tudja, hogy normális (és feltételezhető, hogy egyformák is lettek a lefordított függvények), és ezért csak az egyik kerül be végül a programba.

Egy matematikai példa: polinomok

- Mivel a polinomgyűrűk sok tekintetben tetszőleges test felett ugyanúgy viselkednek, ezért az implementációnál érdemes a testet is paraméterként kezelni.
- A test tehát egy típus lesz, ami template paraméter.
- A műveletek, az összehasonlítás, a behelyettesítés, a legnagyobb közös osztó számítása stb. tetszőleges test felett ugyanúgy végezhetők el.