Láncolt ábrázolású adatszerkezetek a programozásoktatásban

Horváth, Gy[1]–Pap, Gáborné[2]–Szlávi, P[3]
[1]gyozke@inf.elte.hu, [2]papne@inf.elte.hu, [3]szlavip@elte.hu

Absztrakt. Cikkünkben vázoltunk egy módszert, amely legfontosabb didaktikai céljait így foglalhatjuk össze. 1) Bizonyos, adatokkal kapcsolatos fogalmak tisztázása (az adat cím-tartalom kettőssége, típus és adat viszonya, a típus és operáció szoros kapcsolata, adatok élettartam, adatok létrejöttének, ill. összetett adatok részeinek elérés mechanizmusa stb.). 2) Olyan programfejlesztési környezet készítése, amely kellően biztonságos a memóriában való közvetlen manipulációk során (beépített hibafigyelés, a memóriatartalom követhetősége stb.), és könnyű áttérést biztosít az egyes nyelvek által biztosított hatékony direkt memóriakezelés felé (ideális esetben egyszerű kódolási szabályok definiálása árán). 3) az adatabsztrakció növelése (pl. az adatokhoz való hozzáférés sokféleségének bemutatása által). 4) változatos nyelvi lehetőségek felvillantása.

Mindezek hátterében egy gondosan megszerkesztett, egymásra épülő modulsorozat áll; a legalján egy ún. memóriamodellel (2 változatban). Ezen nyugszik az éppen megvalósítandó adatszerkezet. A mi példánkban egy láncoltan ábrázolt sor. Erre (pontosabban: ilyenekre) építhetjük az eredetileg kitűzött feladat programját. Az aktuális oktatási helyzetet figyelembe véve, kétféle („filozófiájú”) programozási nyelvre dolgoztuk ki módszerünket: a „hagyományos”, imperatív nyelvek kategóriájába sorolható Pascalra, és a tárgyközpontúságot is kellően támogató C++ nyelvre.

Kulcsszavak és kifejezések: adatszerkezetek, láncolás, objektum orientált programozás, moduláris programozás, programozásmódszertan

1. Előhang – fogalmak, célok, eszközök, a módszer

1.1. Láncolás a programozási folyamatban

A tudatos programkészítési folyamatban meghatározó szerepe van az adatokkal kapcsolatos döntéseknek. A döntések egy részét a feladat-/program-specifikáció maga meghatározza, másokat a finomítások definiálása során kell meghoznia a programozónak. E döntések nagyban befolyásolják a program *hatékonyság*át: gyorsaságát, a szükséges memória méretét. Különösen akkor, ha az adatok között *sokaság*-félék[1] is előfordulnak. Jellemzően ilyenek: a tömb, a halmaz, a lista, a verem, a sor, a táblázat, a fa, valamint mindközül a legösszetettebb: a gráf.

A sokaság-félék reprezentálása nem nehéz feladat, ha alapnak a –programozási nyelvek által támogatott– tömböt tekintjük. Amint azonban a programban ilyenekből többre is szükség van, felvetődik új problémaként a memória gazdaságos használatának többlet igénye: *Hogy lehetne csak annyi helyet használni, amennyire az adott pillanatban éppen szükség van? Hogyan lehetne a helyfoglalást dinamikussá tenni? Hogy lehetne az adatszerkezetek össz elemigényét összehangolni, s így minimalizálni?*

A dinamizmusra bizonyos nyelvek kínálnak lehetőséget. Az oktatásban használt nyelveknél maradva, kimondhatjuk, hogy a Free Pascalban ([4]) –a Standard Pascaltól ([1,2,3]) eltérően– van *dinamikus* tömb fogalom, a C++ ([5,8]) többféle dinamizálási lehetőséget kínál.

A dinamikus tömb azonban nem ad kézenfekvő választ a korábban felvetett kérdések mindegyikére. Így hozható szóba a másik út: a *láncolt ábrázolás*ra történő áttérés. Ennek lényege az, hogy az adatok *címét* és *tartalmát* fogalmi és kezelési szinten egyaránt *elválasztjuk egymástól*. Az ábrázolás is és a „kezelőszervek” is mutatni fogják ezt a kettősséget. Lesznek az aktuális adat címét, és természetesen tartalmát tartalmazó információk. Lesznek a *cím- és a tartalom kezelésére vonatkozó műveletek*.

1.2. Láncolás a programozásoktatásban

A programozásoktatásban szólni kell az adatokról is, hiszen éppen ezek azok a fogalmak, amelyekkel a feladatmegfogalmazásban elsőként találkozunk. A feladat megoldása során is lépten-nyomon az adatok különféle vonására

---

[1] A sokaság az az adatszerkezet, amelyre jellemző, hogy csupa *azonos típusú elem*ből áll.

építjük algoritmusainkat. Tehát nagyon fontos, hogy az *adatokhoz kapcsolódó fogalmak*kal a leendő programozó tisztában legyen.

Elsők között szokták említeni a *hozzáférési jog*, a *típus* fogalmakat. Tapasztalatunk szerint a következő kérdések homályban szoktak maradni: *Mikor, hogyan jönnek létre a deklarált adatok, meddig léteznek? Mi az adat memóriacíme és mi a tartalma? Mi a típus és az adott típusú adat viszonya?* (Sőt maga a kérdés ilyen formán általában fel sem vetődik, hanem e két fogalom keveredése tapasztalható!) *Mi történik egy tömb indexelésekor, és mi egy rekord (struktúra) mezőjére hivatkozáskor? Hogy történik a paraméterek átadása egy eljárás/függvény hívásakor? ...*

Az ilyen kérdésekre a *láncolás* fogalmának bevezetésével és használatával jó eséllyel választ lehet adni. A cím és tartalom természetes módon válik ketté: a tartalom a címre épülő indirekció útján érhető el. A cím eddig rejtőzködő fogalmának a közelébe jutva a programozó bátrabban *operál* a *címmel* –mégha csak fejben is–: megnövelheti pl. egy mező hosszával, s így a mezőkre hivatkozás módjára jön rá, vagy i darab tömbelem hosszával, s így az i-vel odébb tárolt tömbelem címét állítja elő. Világossá válik, hogy ha egy eljárásba az adatnak a címe jut, az lehetőséget biztosít az értékének a módosítására, ha a tartalma kerül „csak” át, akkor a tartalom esetleges módosulása nem hat vissza az átadott változó tartalmára. … Érdemes a fenti kérdéseket megvizsgálni, miben játszhat tisztázó szerepet az adat cím-tartalom kettősének elválasztása. [10]

1.3. A mutató típus és a láncolás

A láncolást (bármilyen mechanizmussal is valósítjuk meg) az alábbi „szókincsű” ún. *memóriamodell*re vezetjük vissza. Mindössze egy típust, egy konstanst és 3+1 műveletet tartalmaz e nyelv. E memóriamodell „interface-ét” foglaljuk össze az 1. ábrában, magyarázat nélkül használva a [6,7]-ban definiált jelöléseket; viszont az egyes operációk funkcióját kommentben mellékeljük.

```
ExportModul Memória:
  Típus TMemóriaCím   [most elegendő a fogalom létéről tudni, definiálás nélkül]
  Konstans Sehova:TMemóriaCím  [most elegendő a típusát tudni, érték nélkül]
  Eljárás Lefoglal(Változó mut:TMemóriaCím, Konstans db:Egész)
    [a memóriában db bájtnyi egybefüggő helyet foglal le,
     a kezdőcímet mut-ba adja vissza, ha nem sikerült, akkor Sehova-t]
  Eljárás Felszabadít(Változó mut:TMemóriaCím, Konstans db:Egész)
    [a mut-nál kezdődő db bájtnyi egybefüggő memóriát szabadít fel,
     mut-ba Sehova-t tesz]
  Eljárás Értékmásolás(Konstans db:Egész, mut1,mut2:TMemóriaCím)
    [a mut1 címre másol db bájtnyit a mut2 címtől kezdődően]
  Eljárás MemDump
    [Nyomkövetést támogató szolgáltatás: a memória tartalmát megjeleníti]
Modul vége.
```

1. ábra. A memóriamodell exportmodulja.

Ezzel a nyelvezettel már olyan „alapfogalomnak” tekintett összetett adatszerkezetet is le tudunk írni, mint a *vektor* vagy a *mátrix*, sőt a még alapvetőbb *mutató* típust is. A 2. ábrában a mutató típuskonstrukciót[2] definiáljuk.

Ahhoz, hogy futásra kész programhoz jussunk a modellt meg kell valósítani. Ezt kétféleképpen is megtehetjük:

1. „*valódi címekkel*” a *memóriában*, azaz a láncolás egy memóriacím által jön létre (ekkor építünk a programozási nyelvi környezet memóriamenedzselő támogatására) – l. 3a. ábrát;
2. *indexekkel*, amelyek egy kellő méretű (statikusan létrehozott) *tömb*be mutatnak – l. 3b. ábrát.

Vegyük észre, hogy a másodikként javasolt megvalósítás rendelkezik bizonyos „naivitással”: csak *egyféle* típusú (`TElem`: a háttérben meghúzódó tömb elemtípusával azonos típusú) adatok kezeléséhez nyújt segédkezet. Igaz, nekünk most ez tökéletesen elegendő lesz. Az elvárt használat a fenti nyelvet továbbegyszerűsíti. Ugyanis a db paramétert elhagyhatjuk, hiszen a kezelendő adatok csak `TElem` típusúak lehetnek, aminek a hosszát eleve ismerjük (`Méret'TElem`[3]). Ezt a *naiv memóriamodell* nyelvét most nem definiáljuk az 1. ábrán tisztázott modellhez való közelsége miatt, viszont magát a megvalósítás egy jellegzetes részletét a 4. ábrában megadjuk.

---

[2] A „típuskonstrukció” elnevezés arra utal, hogy segítségével lehet létrehozni –paraméterétől függően– különféle típusokat. Látni fogjuk, hogy a mutató típuskonstrukcióval csak egyfajta, nevezetesen az adott típussal megegyező típusú adatokra leszünk képesek mutatni.

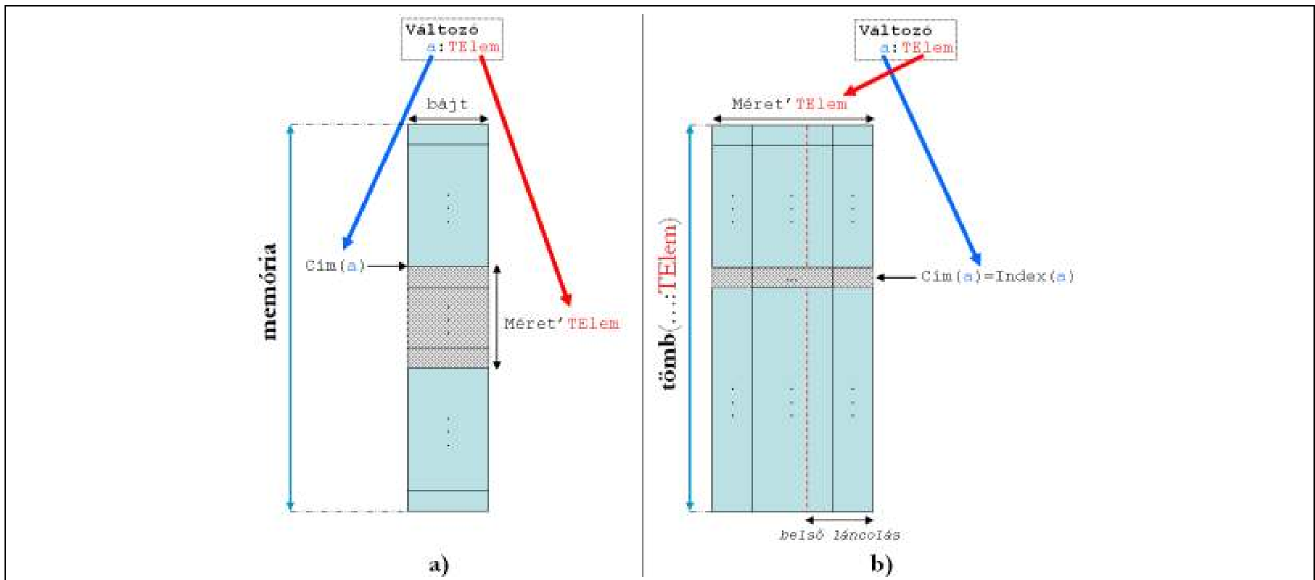
[3] A `Méret'T` operátor tetszőleges T típusú adat helyfoglalását adja meg (bájtokban).

```
Modul Mutató(Típus TElem):
  Másként TElem'Mutató                      [ez a deklarációs minta]
Reprezentáció
  Import: Memória            [az „alap” memóriamodellre építünk: nincs paraméter!]
  Változó kcím:TMemóriaCím                  [a mutató által kijelölt TElem kezdőcíme]
Implementáció
  Eljárás Létrehoz(Változó m:Mutató):
    Lefoglal(kcím, Méret'TElem)
  Eljárás vége.
  Operátor ElemÉrték(Konstans m:Mutató):TElem
    Másként TElem(m)                        [ez az értékhivatkozási minta]
    Értékmásolás(Méret'TElem, ElemÉrték, m)
  Operátor vége.
  Operátor ElemÉrtékadás(Konstans m:Mutató, e:TElem):
    Másként TElem(m):=e                     [ez az értékadási minta]
    Értékmásolás(Méret'TElem, m, e)
  Operátor vége.
Inicializálás
  kcím:=Sehova
Modul vége.
```

2. ábra. A mutató típust megvalósító modul egy részlete.

Megjegyezzük, hogy a `kcím` lesz minden `TElem'Mutató` típusú adat belső ábrázolása, persze egyedileg. ([6,7])

A tömb modulban ([7]) szereplő `Létrehoz` művelettel a napi programozói gyakorlatban nem szoktunk találkozni. Talán éppen ez magyarázza az adatok létrejötte körüli homályt. Ennek megfogalmazási kényszere, logikusan veti fel a kérdést: mikor is kerül erre a vezérlés. És a választ: az adott deklaráció „végrehajtódásakor”. (Ez a Pascal nyelv esetén valóban észrevétlenül történik, nem úgy az adat-deklarációk terén „szabadosabb” C-szerű nyelveknél.)



3a. ábra. A memóriamodell 1. változata: egy `TElem` típusú adat a memóriában.

3b. ábra. A memóriamodell 2. változata: egy `TElem` típusú adat a (memóriahelyettesítő) tömbben.

```
Modul Memória(Típus TElem):
  [Ef: Méret'TElem≥Méret'TMemóriaCím]
Reprezentáció
  Konstans MaxMem:Egész(???)  [a memória méretét rögzítő „belső” paraméter]
  Típus TMemóriaCím=0..MaxMem [⊂ Egész, az Egész műveleteit használni fogjuk]
  Konstans Sehova:TMemóriaCím(0)
  Változó Mem=Tömb(TMemóriaCím:TElem)       [a memória maga; 0. elemet nem használjuk]
Implementáció
  Típus TMaradék=Tömb(1..Méret'TElem-Méret'TMemóriaCím:Byte) ⁴  [a belső láncoláson
                                                                  túli rész]
        TlElem=Rekord(érték:TMaradék, köv:TMemóriaCím)
  Változó bMem:Tömb(TMemóriaCím:TlElem)=Mem       [a memória belső ábrázolása,
                                                   Mem-mel azonos címen]
          szabad,i:TMemóriaCím                    [a szabad elemek listájának feje]
  Eljárás Lefoglal(Változó mut:TMemóriaCím)
    [a memóriában TElem-nyi egybefüggő helyet foglal le,
     a kezdőcímet mut-ban adja vissza, ha nem sikerült, akkor Sehova-t]
    Ha szabad≠Sehova akkor
      mut:=szabad; szabad:=bMem(mut).köv
    különben
      mut:=Sehova
    Elágazás vége
  Eljárás vége.
  Eljárás Felszabadít(Változó mut:TMemóriaCím)
    [a mut-nál kezdődő TElem-nyi egybefüggő helyet szabadít fel, mut-ba Sehova-t tesz]
    Ha mut≠Sehova akkor
      bMem(mut).köv:=szabad; szabad:=mut; mut:=Sehova
    Elágazás vége
  Eljárás vége.
  …
Inicializálás
  Ciklus i=1-től MaxMem-1-ig
    bMem(i).köv:=i+1
  Ciklus vége
  bMem(MaxMem).köv:=Sehova;  szabad:=1
Modul vége.
```

4. ábra. A naiv memóriamodellt statikus tömbben megvalósító modul egy jellegzetes részlete.

A modell felhasználásáról tudnunk kell, hogy a kódja (és adatai) programonként (és típusonként) értelemszerűen csak *egy „példányban”* jöhetnek létre, hiszen egyetlen memória áll rendelkezésre általában is, amelyet e modell most sajátos filozófiájával modellez. A modell moduljában található *inicializáló rész is csak egyszer* hajtódik végre futásonként. (Ezt azért hangsúlyozzuk, mert a modulok inicializáló része –definíciónk szerint– a modul által megvalósított típusú összes adatra, azok deklarációjában egyedileg –tehát mindannyiszor– aktivizálódik.)

1.4. Nyelvi vonatkozások

Célunk, hogy olyan módszert vázoljunk, amely a láncolt ábrázolás lényegét bemutatja még olyan nyelvi környezetben is, amelyből –pl. biztonsági megfontolások miatt– hiányzik a memóriakezelés közvetlen módja.

Kétféle hozzáállással is megvalósítjuk azt az alapot, amelyre építhető az adatok láncolt ábrázolása.

1. „klasszikus” nyelvi lehetőségekre korlátozódva (Free Pascalos szókincs)
2. tárgyközéppontú (objektum-orientált) nyelvek osztálydefiníciós lehetőségeit kihasználva (C++, gcc fordító)

[4] A `TMaradék` típust sokféleképpen definiálhatnánk; a lényeg a helyfoglalása: pontosan `Méret'TElem-Méret'TMemóriaCím` bájtnyi legyen.

Az elsőt azért választottuk, mert közelebb áll a középiskolai programozásoktatás általános szintjéhez. A választott példánk –egy típuskonstrukció– elegáns megvalósítása a modul fogalom paraméterezhetőségét megkívánja, amivel a Free Pascal nem rendelkezik, ezért adunk egy olyan megvalósítást is, amelyben ez a képesség megvan.

1.5. A megközelítés módja

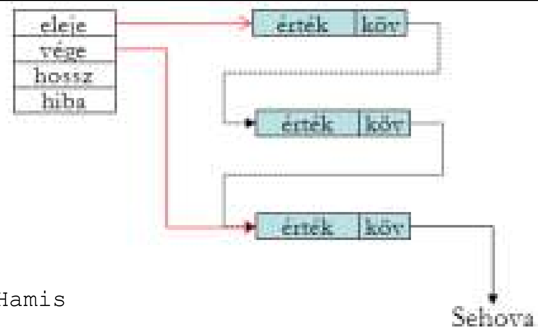
Olyan példát választottunk, amely jól kidomborítja a láncolt ábrázolás lényegét, alaposan „megmozgatja" az adatszerkezetet. A *sor típuskonstrukció*t [6] találtuk ilyennek: eléggé közismert, de kellően változatos műveletekkel rendelkezik. Először megadjuk mint típuskonstrukciós eszközt, definiálva a hozzátartozó műveleteket. (5. ábra)

```
ExportModul Sor(Típus TElem):
  Eljárás Üres(Változó s:Sor)
  Függvény Üres?(Konstans s:Sor): Logikai
  Függvény Tele?(Konstans s:Sor): Logikai
  Függvény Első(Változó s:Sor): TElem
  Eljárás Sorba(Változó s:Sor, Konstans e:TElem)
  Eljárás Sorból(Változó s:Sor, e:TElem)
  Függvény SorHossz(Konstans s:Sor): Egész
  Függvény Hibás?(Változó s:Sor): Logikai
Modul vége.
```

5. ábra. A `Sor` „kezelőszerveinek" definiálása. ([6])

Majd elkészítjük a *láncolt ábrázolású implementáció*ját. Ennek egy jellemző részletét a 6. ábrában foglaljuk össze Mint látható: felhasználunk egy *absztrakt memória modell*t, amely elfedi a láncolás –ebben a pillanatban még– lényegtelen részletkérdéseit. A hangsúly a cím és tartalom kettéválasztásán van.

```
Modul Sor(Típus TElem):
Reprezentáció
  Típus SorElem=Rekord(érték:TElem
                      köv:SorElem'Mutató)
  Változó  eleje,vége:SorElem'Mutató
           hossz:Egész
           hiba:Logikai
Implementáció
  Eljárás Üres(Változó s:Sor):
    eleje:=Sehova; vége:=Sehova; hossz:=0; hiba:=Hamis
  Eljárás vége.
  …
  Eljárás Sorba(Változó s:Sor, Konstans e:TElem):
    Változó
      új:SorElem'Mutató
    Létrehoz(új)
    Ha új≠Sehova akkor
      SorElem(új):=SorElem(e,Sehova)
      Ha vége≠Sehova akkor  SorElem(vége).köv:=új
                 különben eleje:=új
      vége:=új
      hossz:+1
    különben
      hiba:=Igaz
    Elágazás vége
  Eljárás vége.
  …
Modul vége.
```



6. ábra. A Sor láncolt megvalósítása. ([6])

Megjegyezzük, hogy a reprezentációs részben található változók minden sornak egyedileg lesznek részei. Úgy is tekinthetjük, mintha a sor *rekord* adott nevű *mezői* lennének, vagy úgy, mintha egy *objektum adattagjai* lennének. E filozófiát követve hivatkozunk rájuk a műveletek törzsében: következetesen elhagyva az `s` sor-paramétert.

2. Út az „elmélettől" a „gyakorlat" felé

Az oktatásban egy új típuskonstrukció bevezetésekor mindig adottnak (paraméternek) tekintjük azt az elemtípust, amelyből felépítjük a típuskonstrukció segítségével az új típusunkat. Ezt tettük az 5. ábrán látható Sor típuskonstrukció exportmoduljában is. Nevezzük ezt az elemtípust a továbbiakban TElem-nek. A láncolt ábrázolásban az elemek fizikai sorrendje különbözik a logikai sorrendjüktől, pontosan a logikai sorrendet definiáljuk a láncolással. Éppen ezért minden elemet kiegészítünk egy rákövetkezési mutatóval. E mutató típusa ettől kezdve mindig egyértelműen a TElem típusa által meghatározott. Az előző fejezetben bevezetett memóriamodell az elemet és mutatóját tekintette egy egységnek (TElem-nek), hiszen a memória tényleges (fizikai) kezelése szempontjából csak együtt végzünk műveleteket velük (Lefoglal, Felszabadít, Értékmásolás).

Egy típuskonstrukció műveleteinek definiálásakor viszont szükség lehet ezek külön elérésére is. Például, ha egy elem értékét szeretnénk módosítani, akkor csak az elem rész elérésére van szükségünk, ha pedig az értékeket nem módosítjuk, csak a sorrendjük változik, akkor elegendő a mutató rész elérése. A gyakorlatban ennek megfelelően bevezetjük a *láncelem* fogalmát, ami együtt jelenti az adatot a mutatójával, de lehetőséget biztosít ezek külön elérésére is. Másrészt ezen a szinten már nem konkrét memória címekkel foglalkozunk, hanem mutatókon keresztül érjük el a memória bájtjait. Ezek figyelembe vételével fogalmazzuk *újra* a gyakorlatban használt memóriamodell exportmodulját. (7. ábra)

```
ExportModul Memória(Típus TElem):
  Típus TMutató=TLáncElem'Mutató⁵
        TLáncElem=Rekord(érték:TElem, mut:TMutató)
  Konstans Sehova:TMutató  [most még elegendő a típusát tudni, érték nélkül]
  Eljárás Lefoglal(Változó mut:TMutató)
    [a memóriában TLáncElem-nyi egybefüggő helyet foglal le,
     a kezdőcímet mut-ban adja vissza, ha nem sikerült, akkor Sehova-t]
  Eljárás Felszabadít(Változó mut:TMutató)
    [a mut címen kezdődő TLáncElem-nyi egybefüggő memóriát szabadít fel,
     mut-ba Sehova-t tesz]
  Függvény Láncelem(Konstans mut:TMutató):TLáncElem
    [a mut címen kezdődő TLáncElem értékét adja vissza]
  Eljárás Láncelemmódosít(Konstans mut:TMutató, Konstans e:TLáncElem)
    [a mut címen kezdődő TLáncElem értékét módosítja e-re]
  Eljárás Láncelemértékmódosít(Konstans mut:TMutató, Konstans e:TElem)
    [a mut címen kezdődő TLáncElem érték mezőjét módosítja e-re]
  Eljárás Láncelemmutmódosít(Konstans mut:TMutató, Konstans p:TMutató)
    [a mut címen kezdődő TLáncElem mut mezőjét módosítja p-re]
Modul vége.
```

7. ábra. A memóriamodell exportmodulja.

Vegyük észre, hogy az utolsó két művelet nem tartozna a minimális műveletkészlethez, mégis definiáljuk, mert segítségükkel kényelmesebbé válik egy új típuskonstrukció műveleteinek implementálása.

A memóriamodell ilyen irányú megközelítése a korábbi fejezetekben (1.3 és 1.5 fejezet) tárgyalt modell elkülönülő elemeit egységesen, „összemosva" tartalmazza. Ott a memóriamodell csupán az adatok lefoglalását, felszabadítását és másolását tette lehetővé, ezen a szinten különböztetve meg a dinamikus memóriakezelésre épülőt a tömböt használó statikustól. Erre a nagyon memória közeli eszköztárra épült a Mutató típus, amely már kényelmesebb eszközöket biztosított az adatokkal kapcsolatos memóriaműveletekre. A 7. ábrán bemutatott „gyakorlati" memóriamodell a fent elkülönült részeket összevonva tartalmazza. Egyszerre jelenik meg benne a memóriamodell és a mutató típus a TMutató által meghatározott memóriacím típusában, valamint a memóriakezeléshez kapcsolódó műveletekben (Lefoglal, Felszabadít). Itt kerül meghatározásra a láncoláshoz szükséges láncelem típusa is (TLáncElem), valamint a láncelemhez kapcsolódó műveletek (Láncelem, Láncelem-módosít) és kiegészítésképpen két hatékonyságnövelő művelet (Láncelemértékmódosít, Láncelemmutmódosít). Gyakorlatilag a láncelemhez kapcsolódó memóriatípus-specifikus műveletek jelennek meg benne. Erre épülnek a további láncolt adatszerkezetek.

Készítsük most akkor el a Sor típuskonstrukció *láncolt ábrázolású implementáció*ját felhasználva az előző *memória modell*ünket. Mivel az utóbbi memóriamodell típusait és műveleteit használjuk az implementációban importálnunk kell a Memória modult a TElem-mel paraméterezve. (8. ábra)

[5] A választott memóriamodellt elsősorban ez határozza meg.

```
Modul Sor(Típus TElem):
Reprezentáció
  Import:  Memória(TElem)
  Változó  eleje,vége:TMutató
           hossz:Egész
           hiba:Logikai
Implementáció
  Eljárás Üres(Változó s:Sor):
    eleje:=Sehova; vége:=Sehova; hossz:=0; hiba:=Hamis
  Eljárás vége.
  Függvény Üres?(Konstans s:Sor):Logikai
    Üres?:=(hossz=0)
  Függvény vége.
  Függvény Tele?(Változó s:Sor):Logikai
    Változó  p:TMutató
    Lefoglal(p)
    Ha p=Sehova akkor  Tele?:=Igaz
              különben Tele?:=Hamis; Felszabadit(p)
  Függvény vége.
  Függvény Első(Változó s:Sor):TElem
    Ha Üres? akkor  hiba:=Igaz
           különben Első:=Láncelem(eleje).érték
    Elágazás vége
  Függvény vége.
  Eljárás Sorba(Változó s:Sor, Konstans e:TElem):
    Változó új:TMutató
            le:TLáncelem
            se:TElem
    Lefoglal(új)
    Ha új≠Sehova akkor
      le:=TLáncElem(e,Sehova)
      Láncelemmódosít(új,le)
      Ha vége≠Sehova akkor  Láncelemmutmódosít(vége,új)
                  különben eleje:=új
      vége:=új; hossz:=hossz+1
    különben
      hiba:=Igaz
    Elágazás vége
  Eljárás vége.
  Eljárás Sorból(Változó s:Sor, e:TElem)
    Változó p:TMutató
            le:TLáncelem
    Ha Üres?(s) akkor  hiba:=Igaz
              különben le:=Láncelem(eleje)
                       e:=le.érték;  p:=le.mut
                       Felszabadit(eleje)
                       Ha p=Sehova akkor vége:=Sehova
                       eleje:=p;  hossz:=hossz-1
    Elágazás vége
  Eljárás vége.
  Függvény SorHossz(Konstans s:Sor):Egész
    SorHossz:=hossz
  Függvény vége.
  Függvény Hibás?(Változó s:Sor):Logikai
    Hibás?:=hiba; hiba:=Hamis
  Függvény vége.
Modul vége.
```

8. ábra. A Sor láncolt megvalósítása. ([6])

3. Megvalósítás „klasszikus” nyelvi környezetben

„Klasszikus” programozási nyelvnek tekintjük a strukturált programozás oktatására és számos területen való alkalmazására szolgáló Pascal programozási nyelvet. Ennek bármely verzióját vesszük alapul, elmondható róla, hogy nem alkalmasak moduljai a típussal való paraméterezésre.[6] A fentiekben vázolt memóriamodellünk ezért több modulban valósítható meg, melyek mindegyike egy-egy típust definiál. Ebben a fejezetben a megvalósításhoz a Free Pascal nyelvet használjuk, ebben megírt kódokkal szemléltetjük a modell alkalmazásának menetét.

Alapmodulunk lesz a `TElem` típust definiáló (most csak a minimálisan szükséges „szókinccsel” létrehozott) unit, ami aztán minden más modul importlistáján szerepel. (9. ábra)

```
Unit uelem;
Interface
  Type TElem = Integer;
Implementation
Begin
End.
```

Mindig a feladatban szereplő típusra cserélendő, akár elemi, akár összetett típus lehet

9. ábra. A `TElem` típus unitja.

Ezt követi a gyakorlatban használt memóriamodell 7. ábrán *újra fogalmazott* exportmoduljának megvalósítása. Itt már két utat járhatunk be. Választhatjuk

- a memóriamodell 1. változatát (3a. ábra: egy `TElem` típusú adat a memóriában), illetve
- a memóriamodell 2. változatát (3b. ábra: egy `TElem` típusú adat a –memóriahelyettesítő– tömbben).

A továbbiakban a 3a modellt *dinamikus memóriamodell*, a 3b-t pedig *statikus memóriamodell* elnevezéssel fogjuk illetni.

A 10. ábra a dinamikus modell megvalósítását mutatja be, ahol importáljuk az `uelem` unitot, ezzel megvalósítva a `TElem`-mel történő paraméterezést.

```
Unit dinlanc;
Interface
  Uses uelem;
  Type Mutato = ^TLancElem;
       TLancElem = Record
         ertek: TElem;
         mut: Mutato;
       End;
  Const sehova = nil;
  Procedure Lefoglal(Var p: Mutato);
  Procedure Felszabadit(Var p: Mutato);
  Function LancElem(Const p: Mutato): TLancElem;
  Procedure LancElemModosit(Const p: Mutato; Const le: TLancElem);
  Procedure LancElemErtekModosit(Const p: Mutato; Const e: TElem);
  Procedure LancElemMutModosit(Const p: Mutato; Const m: Mutato);
Implementation
  Procedure Lefoglal(Var p: Mutato);
  Begin
    new(p);
  End;
  Procedure Felszabadit(Var p: Mutato);
  Begin
    dispose(p);  p := sehova;
  End;
  Function LancElem(Const p: Mutato): TLancElem;
  Begin
    LancElem := p^;
  End;
```

[6] Pontosabban a Free Pascalban már létezik típussal való paraméterezés, de csak osztály szinten, függvény szinten nem.

```
  Procedure LancElemModosit(Const p: Mutato; Const le: TLancElem);
  Begin
    p^ := le;
  End;
  Procedure LancElemErtekModosit(Const p: Mutato; Const e: TElem);
  Begin
    p^.ertek := e;
  End;
  Procedure LancElemMutModosit(Const p: Mutato; Const m: Mutato);
  Begin
    p^.mut := m;
  End;
Begin
End.
```

10. ábra. A dinamikus memóriamodell unitja.

Hasonló módon valósítjuk meg a statikus modellt is (11. ábra), ahol szintén importáljuk az `uelem` unitot, ezzel megvalósítva a `TElem`-mel történő paraméterezést.

```
Unit statlanc;
Interface
  uses uelem;
  const MaxMem = 100;
  type Mutato = 0..MaxMem;
       TLancElem = Record
          ertek: TElem;
          mut: Mutato;
       end;
       TMem = Array[1..MaxMem] of TLancElem;
  Const sehova = 0;
  Var szfej: Mutato;
      mem: TMem;
      i: integer;
… {Műveletek nevei és paraméterezésük természetesen változatlanok}
Implementation
  Procedure Lefoglal(Var p: Mutato);
  Begin
    p := szfej;
    If p <> sehova then   szfej := mem[szfej].mut;
  End;
  Procedure Felszabadit(Var p: Mutato);
  Begin
    mem[p].mut := szfej;  szfej := p;  p := sehova;
  End;
  Function LancElem(Const p: Mutato): TLancElem;
  Begin
    LancElem := mem[p];
  End;
  Procedure LancElemModosit(Const p: Mutato; Const le: TLancElem);
  Begin
    mem[p] := le;
  End;
  Procedure LancElemErtekModosit(Const p: Mutato; Const e: TElem);
  Begin
    mem[p].ertek := e;
  End;
  Procedure LancElemMutModosit(Const p: Mutato; Const m: Mutato);
  Begin
    mem[p].mut := m;
  End;
Begin
  szfej := 1;
```

Ez az érték a feladattól függően változtatandó, ez a konstans mondja meg, hogy hány `TLáncElem` típusú elemekből álló tömbnek tekintjük a memóriát.

```
    For i:=1 to MaxMem-1 do
    Begin
      mem[i].mut := i+1;
    End;
    mem[MaxMem].mut := sehova;
  End.
```

11. ábra. A statikus memóriamodell unitja.

Nem marad más hátra, mint Pascal nyelven megírni a 8. ábrán látható `Sor` típuskonstrukció *láncolt ábrázolású implementáció*ját. Az implementációban importálnunk kell a választott `memória`modellt és a `TElem` típust definiáló unitot is. (12. ábra)

```
  Unit uslsor;
  Interface
    Uses statlanc, uelem;
    Type TSor=Record
                Eleje {ahonnan kivehetek}: Mutato;
                Vege {ahova tehetek elemet}: Mutato;
                Hossz: Integer;
                Hiba: Boolean;
              End;
  … {Műveletek nevei és paraméterezésük a sor exportmoduljában (8.ábra) definiáltak}
  Implementation
    Procedure Ures(Var S:TSor);
    Begin
      S.Eleje:=sehova;  S.Vege:=sehova;  S.Hossz:=0;  S.Hiba:=False;
    End;
  …
    Procedure Sorba(Var S:TSor; Const E: TElem);
      Var p: Mutato;
          le: TLancElem;
    Begin
      If Tele_e(S) then S.Hiba:=True
      else Begin
        Lefoglal(p);
        le.ertek := e;    le.mut := sehova;
        LancElemModosit(p, le);
        If Ures_e(s) then s.eleje := p
                     else  LancElemMutModosit(s.vege, p);
        s.vege := p; s.hossz:=s.hossz+1;
      End;
    End;
    Procedure Sorbol(Var S:TSor; Var e: TElem);
      Var sv: Mutato;
    Begin
      If Ures_e(S) then S.Hiba:=True
      else begin
        e := LancElem(s.eleje).ertek;  sv := LancElem(s.eleje).mut;
        Felszabadit(s.eleje);
        If sv = sehova then s.vege := sehova;
        s.eleje := sv;  s.hossz:=s.hossz-1;
      End;
    End;
    …
```

Ezt az importot kell dinlanc-ra változtatni, ha a dinamikus memóriamodell unitját szeretnénk alkalmazni.

12. ábra. A statikus memóriamodell unitja.

A fenti megvalósítások esetén még egy dolgot érdemes végig gondolnunk. Mivel programjaink számára felhasználható memóriánk csak egy van, a dinamikus modell alkalmazása esetén a több sort használó programunk ezen a közös területen osztozik. Más kérdés, hogy a típussal való paraméterezés hiánya miatt sajnos, ha egy programon belül több különböző típusú sort szeretnék használni, akkor minden típusnak meg kell feleltetni egy új `Sor` unitot, ami csak a nevében és abban fog különbözni a másik `Sor` unittól, hogy melyik `TElem` típust definiáló unitot importálja. Persze ekkor az egyes műveleteknél a unit nevét, mint „minősítőt” fel kell használnom.

Statikus memóriamodell alkalmazása esetén annyi lesz a különbség, hogy a különböző `TElem` típussal definiált `Sor`-ok különböző memórián (a háttérben rejtőzködő tömbön), míg az azonos `TElem`-űek egy közös memórián osztozkodnak. Ezt az észrevételt a `statlanc` unit `MaxMem` konstansának meghatározásánál fontos figyelembe venni.

4. Megvalósítás tárgyközéppontú nyelvi környezetben

A 2. fejezetben kifejtett memóriamodell (7. ábra) és a Sor típus (8. ábra) tárgyközéppontú nyelven történő implementálása számos különbséget mutat a 3. fejezetben „klasszikus" nyelvi környezetben, Free Pascal nyelven történő megvalósítástól. Ennek egyik oka a tárgyközéppontú programozás filozófiájából fakad, az implementálást befolyásoló másik tényező pedig maga a választott nyelvi környezet. A korszerű tárgyközéppontú nyelvek közül (C++, Java, C#) a C++ nyelvet választottuk GNU C++ fordítóval.

A tárgyközéppontú nyelvek lehetőséget adnak osztályok definiálására, sőt néha megkövetelik azt. Az osztály egyszerre tartalmazza a típus ábrázolásához, működéséhez tartozó adatokat, és a kapcsolódó műveleteket (egységbe zárás). A Pascal megvalósításban a `Sor` modulnak megfelelő egységet a unit biztosította[7]. Azon belül azonban a `Sor` típus ábrázolása és a hozzá tartozó műveletek elkülönülten kerültek definiálásra, kapcsolatot csupán a műveletek első paramétereként megjelenő `TSor` típusú paraméter biztosította. Ráadásul a sor típus reprezentálása (rekord) is a unit interface részébe került, ami így a unitot használó programok számára elérhetővé válik. C++-ban lehetőség van az adattagok és metódusok láthatóságának beállítására, így elérhetjük azt, hogy a sor típusból csupán a műveletek látszódjanak, a konkrét reprezentálás viszont nem.

```
class CSor {
    private:    TMutato eleje, vege;
                int hossz;
                bool hiba;
    public:     CSor();   //Konstruktor
                ~CSor();  //Destruktor
                bool Ures_e();
                bool Tele_e();
                bool Hibas_e();
                int SorHossz();
                void Sorba(const TElem e);
                void Sorbol(TElem &e);
                TElem Elso();
};
```

13. ábra. A `CSor` osztály definíciója.

A 13. ábrán a `CSor` osztály definiálása látható. A reprezentációnak megfelelő adattagok (`eleje`, `vege`, `hossz`, `hiba`) privát mezőkként vannak feltüntetve, ezekhez csak az osztály tagmetódusai férhetnek hozzá. A sor típushoz tartozó műveletek publikusként lettek definiálva, így azokat a `CSor` osztályt példányosító program is használhatja. Formális leírásunkban, illetve Pascalban is szükségünk volt az `Üres` műveletre, amely a sor típust alapállapotba hozta. A tárgyközéppontú nyelvekben ezt a feladatot a konstruktorok végzik el, amelyek az osztály egy példányának létrejöttekor futnak le. A konstruktor neve az osztály nevével egyezik meg, és nincsen visszatérésiérték-típus megjelölve. A konstruktor mellett érdemes felvenni a destruktort is, amely az osztálypéldány megszűnésekor fut le. A sor esetében a destruktorban felszabadíthatjuk a sorműveletek során lefoglalt memóriát. Hasonló módon osztály keretein belül kerültek megvalósításra az egyes memóriamodellek is (l. lentebb).

A sor típus Free Pascal implementációjában annyi példányban kell a sor unitot elkészíteni, ahány féle elemtípussal szeretnénk azt használni. A C++ lehetővé teszi egy osztály definiálásakor a típussal való paraméterezést.

Ezt a nyelv ún. sablonok (template) segítségével éri el[8]. Ezzel megadhatjuk a sor típus definiálásakor a sorban tárolni kívánt elem típusát (`TElem`).

```
template <class TElem>
class CSor {
      ...
}
```

[7] Érdemes megjegyezni, hogy a Free Pascal nyelvben már lehetőség van a C++-ban megismert korszerű osztályok definiálására (class) a régi object mellett.

[8] Más nyelvekben, pl. Javaban vagy C#-ban ezt a lehetőséget a generic-ek biztosítják. Meg kell jegyezni, hogy a C++ sablonjai sokkal több lehetőséget biztosítanak, mint a másik két nyelv generic-jei.

Az osztály definiálásakor megadott sablonparaméter felhasználható az osztályon belül. A műveletek implementálásakor újra fel kell tüntetni azt, hogy sablonnal dolgozunk, valamint innentől kezdve a `CSor` típusra történő bármely hivatkozáskor jelezni kell azt, hogy milyen típusú paraméterrel hívjuk meg.

```
template <class TElem>
void CSor<TElem>::Sorbol(TElem &e) {
      ...
}
```

A sablonok fordítási időben értékelődnek ki a konkrét példányosítás ismeretében, az osztálysablonból annyi különböző osztály jön létre, ahány különböző típusparaméterrel hívtuk meg azokat. Így egy `CSor<int>` és egy `CSor<double>` két különböző osztályt hoz létre.

A sablonok használatának annyi gyakorlati következménye van, hogy a szokásos gyakorlattól eltérően, ahol az osztálydefiníció egy fejléc állományban (.h kiterjesztéssel), a műveletek implementációja pedig egy ugyanolyan nevű, .cpp kiterjesztésű állományban helyezkedik el, a sablonnal definiált osztályok implementációját nem lehet elkülöníteni az osztálydeklarálástól, így azok egyetlen, .h kiterjesztésű állományban foglalnak helyet (pl. sor.h).

Elvárásunk az alkalmazott naiv memóriamodellel szemben (ahogy azt a Free Pascal implementáció meg is valósítja), hogy az azonos elemtípusból felépített sorok ugyanazt a memóriát (statikus láncolás esetén tömböt) használják. Ha a memóriamodell osztályában tagváltozóként jelenne meg a memóriát reprezentáló tömb, akkor az annyi példányban jönne létre, ahány példányban a memóriamodell osztálya létrejön. Mivel ez utóbbit a `CSor` osztályban használjuk a memóriakezelésre, így ahány sor példányunk létrejön annyi „memóriatömb" jönne létre a háttérben.

Erre a problémára a tárgyközéppontú nyelvek az osztályszintű vagy más néven statikus változók és metódusok használatával nyújtanak megoldást. Ezek a változók vagy metódusok az osztály példányosítása nélkül is elérhetőek, és példányonként nem jönnek újra és újra létre. A memóriamodell osztályban statikus láncolás esetén a memóriának megfelelő tömböt statikusként kell tehát létrehozni. Ezt a tömböt azonban inicializálni kell az első használat előtt, hiszen a szabad elemek ekkor kerülnek összefűzésre. A statikus tagváltozók kezdeti beállítására a legtöbb tárgyközéppontú nyelvben a statikus konstruktorok használatosak, a C++ nyelv mi általunk használt fordítója azonban nem ismeri ezt. Ebben az esetben megoldásként a statikus változókat és a típusdefiníciókat egy külön osztályban (csomagolóosztályban) definiáljuk tagváltozókként, és ezen osztály konstruktorában lehetőség nyílik a tagváltozók inicializálására. Abban az osztályban pedig ahol a fenti változókat statikusként szerettük volna használni, a csomagolóosztály egy példányát definiáljuk statikus adattagként. Így a statikus adattagokat a statikus csomagolóosztály adattagjaiként érhetjük el. A 14. ábrán ennek a megoldásnak elvi vázlata látható, ahol a memóriamodell (`MemModell` osztály) statikusnak szánt változói (pl. a „memóriatömb") egy másik csomagolóosztályban vannak definiálva (`Statikus`) és a konstruktorban inicializálva. Az adattagok a `MemModell` osztályban a csomagolóosztály statikus példányán (`sv`) keresztül érhetőek el.

```
class Statikus {
    public:   ...                 //statikusnak szánt adattagok definiálása
              Statikus() {        //konstruktor
                  ...             //statikusnak szánt adattagok inicializálása
              }
};
class MemModell {
    private:   static Statikus sv;  // statikus változók
    ... //további publikus statikus műveletek: Lefoglal, Felszabadít, stb.
};
Statikus MemModell::sv;
```

14. ábra. A `MemModell` osztályba szánt statikus változók definiálása és inicializálása a `Statikus` osztályban.

Mivel a memóriamodell csak statikus adattagokkal dolgozik, így a műveleteket is megfogalmazhatjuk osztályszinten statikus metódusokat használva. Az utolsó két sorban a C++ nyelv elvárása fogalmazódik meg abban, hogy a statikus változóknak az osztálydefiníción kívül adhatunk kezdőértéket, illetve itt kell feltüntetni őket.

A Free Pascalos implementációban annyi sor unitot kellett elkészítenünk, ahány memóriamodellt szerettünk volna használni, hiszen a sor unit `uses` sorában dőlt el, hogy melyik memóriamodellt használjuk. Elvárásként fogalmazhatjuk meg azt, hogy a sor típus példányosításakor határozhassuk meg a felhasználni kívánt memóriamodellt (valahogy így: `CSor<int, StatikusMemóriaModell> s`). Ennek egyik lehetősége, hogy a `CSor` osztályon belül egy paraméteres konstruktort definiálunk, amely a paramétere szerint egy elágazásban a megfelelő memóriamodellt példányosítja. Hátránya e megoldásnak egyrészt az, hogy a példányosító logika a sor osztályon belül helyezkedik el, másrészt az, hogy nem elég rugalmas újabb memóriamodellek bevezetésére. Nagyobb rugalmasságot az biztosítana, ha a `CSor` példányosításakor a `TElem` mellett azt is megadhatnánk típusparaméterként, hogy mely memóriamodellt megvalósító osztályt használja. A megfelelő memóriamodell kiválasztását pedig a fordító döntené el fordítási időben a paraméter szerint.

A fordítási idejű elágazás egyik lehetséges megvalósítása a sablonok parciális specializációja (partial template specialization)[9] ([8,9]). Ennek során felveszünk egy sablonosztályt egy általános sablonparaméterrel, majd ugyanennek az osztálynak elkészítjük egy olyan változatát, ahol a sablonparaméter már valamilyen konkrét típussal vagy értékkel szerepel. Ezzel lehetővé válik az, hogy bizonyos típusú vagy értékű típusparaméterek esetén a sablonosztályunknak más és más implementációját adjuk. Ezt használjuk ki akkor, amikor a memóriamodellünket definiáljuk, ugyanis implementációja jelentősen különbözik a statikus és dinamikus láncolás esetén.

```
enum MemModellTipus {StatikusMemoriaModell, DinamikusMemoriaModell};
template<class TElem, MemModellTipus MM>
class MemModell {};
template<class TElem>
class MemModell<TElem, StatikusMemoriaModell> {
   ... //a statikus memóriamodell típusainak, adattagjainak, műveleteinek definiálása
};
template<class TElem>
class MemModell<TElem, DinamikusMemoriaModell> {
   ... //a dinamikus memóriamodell típusainak, adattagjainak, műveleteinek definiálása
};
```

15. ábra. A memóriamodell különböző implementációjának megadása sablonok részleges specializációjával.

A 15. ábrán látható a sablonok részleges specializációja a memóriamodellünk esetén. Először egy felsorolástípusban megadjuk, hogy hányféle memóriamodell-típusunk van. A `MemModell` sablonosztálynak két sablonparamétere van: az egyik a `TElem`, a másik a memóriamodell-típus. Ez az osztály teljesen kifejtetlen marad. Meg kell viszont fogalmaznunk ezt az osztályt arra az esetre, amikor a második paramétere `StatikusMemoriaModell`, illetve `DinamikusMemoriaModell`. Ezek a specializált osztályok kerülnek kifejtésre, amelyeken belül természetesen más típusú lesz a `Mutató` típus, más a `TLancElem`, és másképpen néznek ki a tagmetódusok implementációi is.

További lehetőségként merül fel a tárgyközéppontú nyelvek esetén a hibák más jellegű kezelése. A formális és a Free Pascal implementációban a műveletek során bekövetkező hibát a hiba változó állapota jelezte. A főprogram csak úgy győződhet meg a végrehajtott művelet hibamentességéről, ha minden sorművelet után lekérdezi a `Hibás_e` művelettel a hibaállapotot. Az ilyen jellegű program hamar áttekinthetetlenné válik a sok felesleges ellenőrzés miatt. C++-ban lehetőség nyílik kivételek kiváltására és kezelésére[10]. Ha egy művelet hibára vezet, akkor egy kivételt dob (`throw`, l. 18. ábra), amit a műveletet meghívó programban kezelni kell. Az olyan programblokkokat, amelyek kivételt dobhatnak, `try` blokkba kell helyeznünk, és a bennük esetlegesen felmerülő kivételeket a `catch` blokkban kezelnünk (l. 19. ábra).

A fentiek alapján a memóriamodell C++ implementációja a 16. és 17. ábrán látható, az előbbiben a statikus, az utóbbiban a dinamikus memóriamodellé. A két ábra tartalma ugyanannak a .h kiterjesztésű állománynak a része. Az olvashatóság érdekében a metódusok implementációjának leírásában a metódusok fejlécét az alábbi egyszerűsítéseknek megfelelően tettük rövidebbé:

```
void Lefoglal(Mutato &p) { ... }              //rövidített változat
template <class TElem>                        //implementáláshoz szükséges változat
void MemModell<TElem, StatikusMemoriaModell>::Lefoglal(Mutato &p) { ... }
```

```
TLancElem LancElem(const Mutato p) { ... }   //rövidített változat
template <class TElem>                        //implementáláshoz szükséges változat
typename MemModell<TElem, StatikusMemoriaModell>::TLancElem
         MemModell<TElem, StatikusMemoriaModell>::LancElem(const Mutato p) { ... }
```

A fenti kódrészlet első egyszerűsítése a void visszatérésiérték-típussal definiált függvények esetében alkalmazható, míg az alsó azoknál a függvényeknél, ahol egy osztálybeli típus a visszatérési érték típusa.

A memóramodellt használó sor típus implementációjának fontosabb részei a 18. ábrán, míg használata a 19. ábrán látható.

[9] Ez egy C++-specifikus elem, a generic-eket használó nyelvek ezt nem ismerik.

[10] Erre a Free Pascalban is lehetőség van

```
enum MemModellTipus {StatikusMemoriaModell, DinamikusMemoriaModell};
template<class TElem, MemModellTipus MM>
class MemModell {};
//--------------------StatLancStatikus-----------------------
template <class TElem>
class StatLancStatikus {
    public:   typedef int Mutato;
              struct TLancElem { TElem ertek; Mutato mut; };
              int MAXMEM;
              Mutato szfej;
              TLancElem* mem;
              Mutato sehova;
              StatLancStatikus() {
                  MAXMEM = 100;
                  mem = new TLancElem[MAXMEM];
                  sehova = -1; szfej=0;
                  for (int i = 0; i<MAXMEM-1; ++i) { mem[i].mut=i+1; }
                  mem[MAXMEM-1].mut=sehova;
              }
};
//--------------------Statikus memoriamodell-----------------
template<class TElem>
class MemModell<TElem, StatikusMemoriaModell> {
    public:   typedef typename StatLancStatikus<TElem>::Mutato Mutato;
              typedef typename StatLancStatikus<TElem>::TLancElem TLancElem;
    private:  static StatLancStatikus<TElem> sv;
    public:   static void Lefoglal(Mutato &p);
              static void Felszabadit(Mutato &p);
              static TLancElem LancElem(const Mutato p);
              static void LancElemModosit(const Mutato p, const TLancElem le);
              static void LancElemErtekModosit(const Mutato p, const TElem e);
              static void LancElemMutModosit(const Mutato p, const Mutato m);
              static Mutato sehova();
};
template <class TElem>
StatLancStatikus<TElem> MemModell<TElem, StatikusMemoriaModell>::sv;
void Lefoglal(Mutato &p) {
    p = sv.szfej;
    if (p!=sv.sehova) { sv.szfej = sv.mem[sv.szfej].mut; }
}
void Felszabadit(Mutato &p) {
    sv.mem[p].mut = sv.szfej;   sv.szfej = p;   p = sv.sehova;
}
Mutato sehova() {
    return sv.sehova;
}
TLancElem LancElem(const Mutato p) {
    return sv.mem[p];
}
void LancElemModosit(const Mutato p, const TLancElem le) {
    sv.mem[p] = le;
}
void LancElemErtekModosit(const Mutato p, const TElem e) {
    sv.mem[p].ertek = e;
}
void LancElemMutModosit(const Mutato p, const Mutato m) {
    sv.mem[p].mut = m;
}
```

16. ábra. A statikus memóriamodell C++ implementációja (memmodell.h).

```
//--------------------DinLancStatikus----------------------
template <class TElem>
class DinLancStatikus {
    public:   struct TLancElem;
              typedef TLancElem* Mutato;
              struct TLancElem {
                  TElem ertek;
                  Mutato mut;
              };
              Mutato sehova;
              DinLancStatikus() {
                  sehova = NULL;
              }
};
//--------------------Dinamikus memoria modell-------------
template<class TElem>
class MemModell<TElem, DinamikusMemoriaModell> {
    public:   typedef typename DinLancStatikus<TElem>::Mutato Mutato;
              typedef typename DinLancStatikus<TElem>::TLancElem TLancElem;
    private:  static DinLancStatikus<TElem> sv;
    public:   static void Lefoglal(Mutato &p);
              static void Felszabadit(Mutato &p);
              static TLancElem LancElem(const Mutato p);
              static void LancElemModosit(const Mutato p, const TLancElem le);
              static void LancElemErtekModosit(const Mutato p, const TElem e);
              static void LancElemMutModosit(const Mutato p, const Mutato m);
              static Mutato sehova();
};
template <class TElem>
DinLancStatikus<TElem> MemModell<TElem, DinamikusMemoriaModell>::sv;
void Lefoglal(Mutato &p) {
    p = new TLancElem;
}
void Felszabadit(Mutato &p) {
    delete p;   p = 0;
}
Mutato sehova() {
    return MemModell<TElem, DinamikusMemoriaModell>::sv.sehova;
}
TLancElem LancElem(const Mutato p) {
    return *p;
}
void LancElemModosit(const Mutato p, const TLancElem le) {
    *p = le;
}
void LancElemErtekModosit(const Mutato p, const TElem e) {
    (*p).ertek = e;
}
void LancElemMutModosit(const Mutato p, const Mutato m) {
    (*p).mut = m;
}
```

17. ábra. A dinamikus memóriamodell C++ implementációja (memmodell.h).

```
#include "memmodell.h"
template <class TElem, MemModellTipus MMT >
class CSor {
    private:   typedef MemModell<TElem, MMT> MM;
               typedef typename MM::Mutato TMutato;
               typedef typename MM::TLancElem TLancElem;
    public:    enum HibaTipus {SorTele, SorUres};
    // Az 14. ábrán bemutatott privát adattagok és publikus metódusok
};
... //Többi művelet kifejtése
template <class TElem, MemModellTipus MMT>
void CSor<TElem, MMT>::Sorba(const TElem e) throw (HibaTipus) {
    if (Tele_e()) { hiba = true;  throw SorTele; }
    else {
        TMutato p;  TLancElem le;
        MM::Lefoglal(p);
        le.ertek = e;  le.mut = MM::sehova();
        MM::LancElemModosit(p, le);
        if (vege != MM::sehova()) {
            MM::LancElemMutModosit(vege, p);
        } else { eleje = p; }
        vege = p; hossz += 1;
    }
}
template <class TElem, MemModellTipus MMT>
void CSor<TElem, MMT>::Sorbol(TElem& e) throw (HibaTipus) {
    if (Ures_e()) { hiba = true;  throw SorUres; }
    else {
        e = MM::LancElem(eleje).ertek;
        TMutato p = MM::LancElem(eleje).mut;
        MM::Felszabadit(eleje);
        if (p == MM::sehova()) { vege = MM::sehova(); }
        eleje = p; hossz -= 1;
    }
}
```

18. ábra. A sor típus C++ implementációjának fontosabb részei (sor.h).

```
#include <iostream>
#include "memmodell.h"
#include "sor.h"
using namespace std;
typedef CSor<int, StatikusMemoriaModell> CSorIntStat;
int main() {
    CSorIntStat s;
    try {
        for (int i = 0; i<10; i++) { s.Sorba(i); }
        cout << "Elso: " << s.Elso() << endl;
        for (;!s.Ures_e();) { cout << s.Sorbol() << endl; }
    }
    catch (CSorIntStat::HibaTipus h) {
        switch (h) {
            case CSorIntStat::SorTele: {
                cout << "Megtelt a sor" << endl;
                break;
            }
            case CSorIntStat::SorUres: {
                cout << "Ures a sor" << endl;
                break;
            }
        }
    }
    return 0;
}
```

19. ábra. A sor típus használata.

5. Utóhang – visszatekintés, következtetések

Cikkünkben vázoltunk egy módszert, amelynek legfontosabb didaktikai céljai a következők voltak:

1. bizonyos, adatokkal kapcsolatos fogalmak tisztázása (az adat cím-tartalom kettőssége, típus és adat viszonya, a típus és operáció szoros kapcsolata, adatok élettartam, adatok létrejöttének, ill. összetett adatok részeinek elérés mechanizmusa stb.),
2. olyan programfejlesztési környezet készítése, amely kellően biztonságos a memóriában való közvetlen manipulációk során (beépített hibafigyelés, a memóriatartalom követhetősége stb.), és könnyű áttérést biztosít az egyes nyelvek által biztosított hatékony direkt memóriakezelés felé (ideális esetben egyszerű kódolási szabályok definiálása árán),
3. az adatabsztrakció növelése (pl. az adatokhoz való hozzáférés sokféleségének bemutatása által),
4. változatos nyelvi lehetőségek felvillantása.

Elsősorban a felsőfokú informatikaoktatásban látjuk a helyét egy programozással, adatszerkezetekkel foglalkozó tárgyban. Meg kell említenük, hogy lényegében ezt a módszert követtük egy középiskolásoknak szervezett programozási versenyre felkészítő tanfolyam során is. ([11]) Itt megelégedtünk egyetlen nyelvi környezet (a Turbo Pascal) adta szolgáltatásokkal, de tisztán megtartottuk a memóriamodell és a ráépülő adatszerkezetek (az ún. hiányos mátrix és a lista) függetlenségét, azaz önálló unitokban különítettük el.

A cikkünkben nem tértünk ki olyan módszertani kérdésre, hogy mindezekből mely algoritmikus és kódrészeket kell elkészítenie a hallgatóságnak magának, s –a tudnivalók megismerése után– melyeket használhatják fel, mint kész keretet. A tevékenység ezen kettévágásával gyorsítani tudtuk a sikerélményekhez jutást, s így végső soron az oktatás hatékonyságát.

6. Hivatkozásjegyzék

[1] Jensen, K.–Wirth, N.: „*Pascal User Manual and Report*", Springer-Verlag, New York Inc., 1983

[2] „*Pascal ISO 7185:1990*", http://standardpascal.org/iso7185.pdf (in Hungarian)

[3] Temesvári, T.–Szlávi, P.–Zsakó, L.: „*Programozási nyelvek: Alapfogalmak*", ELTE IK, 2004 (in Hungarian)

[4] Van Conneyt, M.: „*Free Pascal: Reference guide*", 2010, ftp://ftp.Free Pascal.org/pub/fpc/docs-pdf/ref.pdf

[5] Illés, Z.: „*Programozási nyelvek: C++*", ELTE IK, 1996 (in Hungarian)

[6] Pap, Gné–Szlávi, P.–Zsakó, L.: „*Módszeres programozás: Adattípusok*", ELTE IK, 1998 (in Hungarian)

[7] Szlávi, P.: „*A tömb típuskonstrukció*", (előadás elektronikus kézirata), 2004, http://people.inf.elte.hu/szlavi/PrM2felev/Pdf/PrMea2_3.pdf (in Hungarian)

[8] Austern, M. H.: „*Generic Programming and the STL: using and extending the C++ Standard Template Library*", Addison-Wesley, 1999

[9] Alexandrescu, A.: „*Modern C++ Design – Generic Programming and Design Patterns Applied*", Addison-Wesley, 2003

[10] Cormen, T. H. et al: „*Introduction to Algorithms*", The MIT Press/McGraw Hill, 2001/2003

[11] Szlávi, P.: „*Memóriamodell (Statikus láncolt tömb)*", 2008 (szakköri anyag elektronikus kézirata), http://people.inf.elte.hu/szlavi/PrM2felev/MemModell/MemModell.pdf (in Hungarian)