AZ OBJEKTUM-ORIENTÁLTSÁG ALAPFOGALMAINAK BEMUTATÁSA A C++ NYELV SEGÍTSÉGÉVEL

Szkiba Iván, *szkiba@math.klte.hu*
Vég Csaba, *vega@dragon.klte.hu*
KLTE Matematikai és Informatikai Intézet
KLTE Informatikai és Számító Központ

The topic of this lecture is how to teach the basic concepts of the Object Oriented Methodology. The two key concepts of the Object Oriented Methodology is the abstraction (or inheritance) and the encapsulation. We would like to show how this works by the means of the C++ programming language.

A Kossuth Lajos Tudományegyetemen az objektum orientált szemlélet bemutatása alapvetõen a C++ programozási nyelv segítségével történik. Erre a nyelvre az alábbi szempontok szerint esett a választás:

- A szakos hallgatók a reguláris képzés keretein belül az elsõ évfolyamokon már megismerték a C programozási nyelvet, így a C++ oktatása során elegendõ az objektum orientált eszközkészletre helyezni a hangsúlyt.
- A nyelv a C programozási nyelv népszerûségébõl adódóan az egyik legelterjedtebb objektum orientált programozási nyelv.
- Hordozható és platformfüggetlen, elérhetõ az Egyetem minden számítógépén, a személyi számítógépektõl az erõforrás gépekig. Fontos szempont, hogy már egy átlagos személyi számítógépen is használható, így a legtöbb hallgató otthon is gyakorolhat.
- A programozási nyelvek közül a C++ nyelv az objektum orientált szemlélet eszközkészletének egyik legteljesebb megvalósítása.
- A mindennapi programozási munka során a hallgatók nagyobb valószínûséggel fognak találkozni a C++ nyelvvel, mint bármely más objektum orientált nyelvvel.

Az objektum orientált szemlélet egyik alapvetõ eszköze az öröklõdés (*inheritance*). Az öröklõdés segítségével az azonos jellegzetességû osztályok közös attribútumai és metódusai kiemelhetõk és egy helyen, egyetlen õsosztályban definiálhatók. A kiemelés lehetõvé teszi a programon belüli redundancia csökkentését. Az öröklõdés bemutatása rendszerint grafikus objektumok példáján keresztül (pont, kör, ellipszis) történik. Az elõadás az öröklõdés oktatásának egy másik lehetséges megközelítését mutatja be, az absztrakt adatszerkezetek implementációin keresztül. A példák lefedik a származtatás fõbb típusait.

A bezárás (*encapsulation*), mint az objektum orientált szemlélet másik fontos jellemzõje kiemelt szerepet kap az oktatásban. A bezárás olyan absztrakt objektumok definiálására ad lehetõséget, melyek kizárólag a megadott protokollon keresztül érhetõk el. A bezárt objektumok belsõ algoritmikus vagy adatreprezentációs okból történõ módosítása nem vonja maga után a program többi részének módosítását. A kis változtatások így nem terjednek tovább. Az absztrakt osztályok segítségével lehetõség van kizárólag a protokoll megadására. Az absztrakt osztályból származtatott konkrét osztályok az implementáció során ezt a keretet töltik fel tartalommal. A módszer felhasználásával a program mennyiségi módon, újabb származtatott osztályok egyszerû hozzáadásával bõvíthetõ. A példák az elõadás elõzõ részével összhangban az absztrakt adatszerkezetek implementációs problémáira épülnek.

1. Öröklés

A valós világ egyes objektumai közös jellemzõkkel rendelkezhetnek. A modellezés során a közös jellemzõk kezelésére szolgáló eszköz az öröklés. A közös jellemzõket kiemelve létrehozható egy absztrakt objektum, amely az illetõ objektumok õseként szolgálhat. Ezek után az objektumok az õs objektumtól öröklik a közös jellemzõket. Természetesen a jellemzõket tekintve egyszerû tartalmazás is elképzelhetõ. Ilyenkor nincs szükség az absztrakt objektumra, ugyanis az egyik objektum közvetlenül örökölheti a másik jellemzõit.

Az öröklés az egyes nyelvekben különbözõképpen jelenik meg. Vannak nyelvek, melyek minden származtatás során csak egy bázis osztály használatát teszik lehetõvé, míg mások (mint pl. a C++ programozási nyelv) megengedik a többágú öröklést is.

Az öröklés során a származtatott osztály örökli a bázis osztály attribútumait, valamint tevékenységeit. Szokás osztályozni az öröklést szerint, hogy mi volt az öröklés elsõdleges célja. Ezek szerint minimálisan az alábbi típusokat szokás megkülönböztetni: helyettesítés (substitution), tartalmazás (inclusion), korlátozás (constraint), valamint specializáció (specialization). Ezen típusok rendszerint egymással keveredve jelennek meg.

1.1. Helyettesítés (substitution)

Helyettesítésrõl akkor szokás beszélni, ha a származtatott osztály több tevékenységgel, mûvelettel rendelkezik mint a bázis osztály, továbbá az is teljesül, hogy a bázis osztály objektumai helyettesíthetõk a származtatott osztály objektumaival. Az öröklés ezen típusának az alapja az objektum viselkedése, nem pedig attribútumai.

1.2. Tartalmazás (inclusion)

Tartalmazásról szokás beszélni, ha az öröklés alapja a struktúra, s a származtatott osztály objektumai egyben objektumai a bázis osztálynak is.

1.3. Korlátozás (constraint)

A korlátozás egy speciális esete a tartalmazásnak. A származtatott osztály minden objektuma egyben objektuma a bázis osztálynak, csak kielégítenek egy plusz, korlátozó feltételt.

Egy lehetséges példa korlátozásra a sor illetve a kétvégû sor implementálása. A kétvégû sor implementálása után ebbõl egy korlátozással származtatható a sor, mégpedig oly módon, hogy a kétvégû sor egyik végén a 'put' másik végén pedig a 'get' mûvelet korlátozandó:

```
class Deque {
public:
   Deque( int aSize=50 );          // Default méret 50 elem
   ~Deque();
   int putFirst(int);              // Elem az elejére
   int putLast(int);               // Elem a végére
   int getFirst();                       // Elsõ elem
   int getLast();                        // Utolsó elem
protected:
   int  resize(int);               // Átméretezés
   int  theSize;                         // Aktuális méret
   int* theQueue;                        // A terület címe
   int  theFirst;                        // Elsõ elem indexe
```

```
   int  theLast;                        // Utolsó elem indexe
};

class Queue: protected Deque {
public:
   Queue( int aSize=50 );          // Default méret 50 elem
   ~Queue();
   int put();                      // Elem elhelyezése
   int get();                      // Elem kivétele
}
```

1.4. Specializáció (specialization)

A specializáció során az osztály rendszerint újabb attribútumokkal bõvül, s a származtatott osztály objektumai több, speciálisabb információk tárolására lesznek alkalmasak.

A specializációra példa lehet egy egész elemekbõl álló sor és rendezett sor implementálása. A sor implementálása után a rendezett sor ebbõl származtatható oly módon, hogy pl. a 'put' tevékenység átdefiniálásával az elemek mindig a rendezettség szerinti helyükre kerülnek:

```
calss Queue {
public:
   Queue( int aSize=50 );          // Default méret 50 elem
   ~Queue();
   virtual int put(int);                // Elem elhelyezése
   int get();                      // Elem kivétele
protected:
int  resize(int);                  // Átméretezés
   int  theSize;                        // Aktuális méret
   int* theQueue;                       // A terület címe
   int  theFirst;                       // Elsõ elem indexe
   int  theLast;                        // Utolsó elem indexe
};

class SortedQueue : public Queue {
public:
   SortedQueue( int aSize=50 );    // Default méret 50 elem
   ~SortedQueue();
   virtual int put(int);                // Elem elhelyezése
}
```

1.5. Késõi kötés

Az örökléshez kapcsolódnak olyan speciális fogalmak, mint túlterhelés, átdefiniálás, valamint a késõi kötés. Ezen fogalmak közül a késõi kötés ismertetésére érdemes külön hangsúlyt fektetni. Késõi kötésrõl akkor szokás beszélni, ha bizonyos tevékenységek esetén csak futási idõben dönthetõ el, hogy mely eljárás használata szükséges.

A késõi kötés szemléltetésére tekintsünk egy számítógép kereskedést, ahol feladat egy alkatrész nyilvántartás elkészítése. Legyen egy Alkatresz nevû bázis osztály, valamint alkatrész típusonként egy-egy származtatott osztály (pl. Memoria, Alaplap stb). A származtatott osztályok örököljenek egy print nevû tevékenységet, mely egy árlista nyomtatáshoz szükséges információkat jeleníti meg az adott objektumról. Tételezzük fel továbbá, hogy az alkatrészek egy láncolt listán helyezkednek el. Az alábbi program részlet megjeleníti az aktuális árlistát:

```
class Alkatresz {
.
.
```

```
.
   void print();
.
.
.
   };

class Memoria : Alkatresz {
.
.
.
   void print();
.
.
.
   };


Alkatresz* p;

for( p=elso; p != NULL; p=p->kovetkezo )
   p->print();
```

Nos, a fenti program részlet késõi kötés, vagyis ami itt azzal egyenértékû, virtuális tevékenységek használata nélkül nem mûködik helyesen, a lista minden eleme esetén a bázis osztály print tevékenysége kerül meghívásra. Ezzel szemben virtuális tevékenységként deklarálva a print tevékenységet, mindig a megfelelõ print tevékenység kerül meghívásra.

```
cass Alkatresz {
.
.
.
   virtual void print();
.
.
.
   };
```

2. Bezárás

2.1. A fogalom

A bezárás (encapsulation) fogalma több szinten is értelmezhetõ.

A bezárás egyrészt jelentheti az objektum-orientáltságnak azt az alapvetõ technikáját, amellyel egyetlen, osztálynak nevezett egységben definiálja az egymással szorosabb kapcsolatban levõ adatelemek szerkezetét és az azokat kezelõ függvényeket (módszereket). Ennek az *egységbezárás*nak is nevezett technika alapvetõ jelentõségét mi sem bizonyítja jobban, mint hogy ennek hiányában egy eszközkészletet nem minõsíthetünk objektum-orientáltnak. Az egységbezárással egy (absztrakt) objektumot adhatunk meg, amely egyetlen egységben definiálja az összetett állapotának ábrázolásához szükséges adatszerkezetet és egyben az ezen állapot lekérdezését (közvetve az objektum hatását) és transzformálását (az objektumra történõ hatást), vagy egyszerre mindkettõt megvalósító módszereket, a mûködést. Az egységbezárás így az adatszerkezetek absztraktabb leírását teszi lehetõvé.

Másrészrõl a *bezárás* egy olyan technika, amellyel bizonyos szorosabban összetartozó adatok és végrehajtások csoportjának pontos *implementáció*ja elrejthetõ és azok mindössze egy *felület*en (interfészen) keresztül érhetõk el. A bezárásnak ez a típusa egy absztrakciós módszer, mivel a felület elrejti a pontos megvalósítást és annak csak egy általánosított képét mutatja meg. Ez a fogalom tehát bizonyos mértékig független az objektum-orientáltságtól (a moduláris programozás és a TurboPascal unit-jai is ezt az elvet követik), de a legtisztábban ebben a szemléletben jelenik meg. Az objektum-orientáltságban már maguk a programok építõelemei, az objektumok is természetes lehetõséget nyújtanak a bezárásra és tapasztalati tény, hogy ha egy leírási mód, egy programozási nyelv nem ad koncepciójába illeszkedõ és kézenfekvõ jelölést egy technikára, akkor azt a módszert az adott nyelven gyakorlatilag senki nem fogja használni. Egy nyelv lehetõségei és korlátai (Dante képét követve) pecsétnyomóként használójának szemléletét is formázzák, közvetve így karakterizálják (a karakter eredetileg pecsétnyomót jelentett) a nyelven megfogalmazott mondatokat, a programozási nyelven megírt programokat.

A legszûkebb értelemben vett bezárásról akkor beszélünk, ha egy jelölési mód nem csak lehetõvé teszi a megvalósítás elrejtését, hanem a felület megadására és magára a bezárásra is külön eszközkészlettel és jelölésekkel rendelkezik. Bármely nyelven programozhatunk úgy, hogy a program egy adott részletét csak egy felületen keresztül érjük el. De egy megfelelõ eszközkészlettel jól láthatóan szétválaszthatjuk az interfészt és a megvalósítást, valamint a jelölés mellett biztosíthatjuk, hogy a programrészlet (az objektum) elérése csak a felületen keresztül történhessen. Ez utóbbi különösen a több embert igénylõ nagy rendszerek megvalósításakor válik lényeges szemponttá.

2.2. Elõnyök

A bezárás technikájának alkalmazásával világosabb és áttekinthetõbb programszerkezeteket készíthetünk.

Ha egy objektum konkrét megvalósítását elrejtjük és az objektum szolgáltatásait csak egy adott felületen érjük el, akkor az implementáció megváltoztatása nem fog kihatni a program más részeire, az objektumot felhasználó más objektumokra. Rumbaugh és társai megfogalmazásában: a kis változtatás nem fog továbbgyûrûzni, továbbterjedni. Egy rendszer egyes részei így továbbfejleszthetõk, általánosíthatók és optimalizálhatók anélkül, hogy módosítanunk kellene más részeket.

Azonos felületet más objektumok különbözõképpen is implementálhatnak. Ebben az esetben a felületet a régebbi megfogalmazás szerint vezérlõnek vagy meghajtónak (driver), az új megfogalmazás szerint pedig *absztrakt osztály*nak nevezzük. Az absztrakt osztály mindössze a felületet, a protokollt definiálja, amelyet az abból származtatott konkrét osztályok töltenek fel implementációs tartalommal. Egy részfeladat esetén az azonos módon viselkedõ objektumok közül a feladathoz leginkább illeszkedõ konkrét objektum megvalósítása választható ki. Egyben egy rendszer az azonos felületet szolgáltató újabb objektumokkal egészíthetõ ki. Tehát a rendszer extenzív, azaz egyszerû mennyiségi módon bõvíthetõ. Az azonos felületet különbözõképpen implementáló objektumok a többalakúság (polimorfizmus) fogalmáig vezetnek el. Többalakúság esetén bizonyos objektumok azonos mûveleteihez más és más konkrét megvalósítás tartozik.

2.3. A példákról

A bezárás tanításakor (mint általában bármely más tanítás során) a fogalmakat célszerû példákon keresztül bemutatni, kihangsúlyozva, hogy az egyes technikák felhasználása milyen közvetlen és gyakorlati hasznot jelenthet valós problémák megoldásakor. Ullmann megfogalmazásában: "a legjobb gyakorlat egy jó elmélet". A példák azonban mindig csak példák maradnak: lehetetlen egy több embert igénylõ, több éven át fejlesztett rendszer esetén felmerülõ problémák fontosságát egy néhány soros példaprogram segítségével bemutatni. Az azonban könnyen elképzelhetõ, hogy egy feladat megvalósítása annál jobb minõségû, minél kevesebb abban a redundancia, azaz a hasonló jellegû részek csak egyszer szerepelhetnek. Redundancia esetén egy részlet megváltoztatása általában a hasonló részek változtatását is implikálja, és ha ez (figyelmetlenségbõl) elmarad, a rendszer inkonzisztenssé válhat. Egy jelölésrendszer esetén a leglényegesebb szempont így az, hogy

koncepcionálisan mennyire teszi lehetõvé a hasonló jellegû részletek kiemelhetõségét (definiálhatók-e benne például template-osztályok), valamint a kiemelt részletre történõ lehetõ legegyszerûbb hivatkozást. Mivel egy részletet csak egyszer definiálunk, de arra általában többször, esetleg több százszor hivatkozunk, ezért a célunk a lehetõ legrövidebb és legegyszerûbb hivatkozás.

A következõ példákban feltételezzük, hogy egy rendszerben valamely inputként beolvasott és letárolt kifejezéseket kell ismételten kiértékelnünk. A kiértékelés egyszerûsítése miatt a kifejezéseket fordított lengyel jelölésre, azaz postfix formára alakítjuk és úgy tároljuk le. A feladat tehát az, hogy postfix kifejezéseket ki tudjunk értékelni. A példákban tudatosan eltekintünk a hibakezeléstõl, azaz feltételezzük, hogy a hibás kifejezéseket már az elemzés fázisa kiszûrte és csak helyes postfix kifejezéseket kell kiértékelnünk.

A postfix kifejezés kiértékeléséhez szükségünk van egy veremre (stack). Elõször készítsünk el egy rövid C programot, amely egy tömbként implementált verembe elmenti az 1 2 3 számokat, majd azokat egyenként a verembõl kiemeli és kiírja.

```
#include <stdio.h>

#define stLEN 40

double st[ stLEN ];  int sp = -1;

void main() {

    st[ ++sp ] = 1; st[ ++sp ] = 2; st[ ++sp ] = 3;

    printf( "%g, ", st[ sp-- ] );
    printf( "%g, ", st[ sp-- ] );
    printf( "%g\n", st[ sp-- ] );
   }
```

A programot ezután kiegészíthetjük egy olyan függvénnyel, amely a verem tetején elhelyezkedõ két elemet leemeli, azon egy kétoperandusú mûveletet hajt végre, majd az eredményt visszahelyezi a verem tetejére. A példában az "1+2*3" kifejezés "1 2 3 * +" postfix alakjának megfelelõen elmentjük az 1 2 3 számot, majd végrehajtjuk a "*" és a "+" mûveletet, végül kiírjuk az eredményt.

```
#include <stdio.h>

#define stLEN 40

double st[ stLEN ];  int sp = -1;

void Op(char c, double st[], int *sp) {
    double y = st[ (*sp)--],  x = st[ (*sp)-- ];

    switch( c ) {
      case '+': x += y; break;
      case '-': x -= y; break;
      case '*': x *= y; break;
      case '/': x /= y; break;
     }
    st[ ++(*sp) ] = x;
   }

void main() {
```

```
    st[ ++sp ] = 1; st[ ++sp ] = 2; st[ ++sp ] = 3;

    Op('*', st, &sp);  Op('+', st, &sp);


    printf( "%g\n", st[ sp-- ] );
   }
```

Mint látjuk, a programban gyakorlatilag alig történt kiemelés, ezért a veremmûveletek és az operátor hívása csak körülményesen oldható meg. Valós feladat esetén ez a körülményesség különösen akkor válik lényegessé, ha egyszer az implementációt meg szeretnénk változtatni, például a verem megvalósítását változó méretû tömbre, vagy láncolt listára akarjuk lecserélni. Ekkor ugyanis a programban az összes veremmel kapcsolatos mûveletet módosítanunk kell. Gyakorlatilag szintén lehetetlen a programban különbözõ verem-implementációkat egymás mellett szerepeltetni és a részfeladathoz a legjobban illeszkedõt kiválasztani, majd azt a többi megvalósítással azonos módon használni.

Az objektum-orientáltság megközelítésében egyetlen egységben definiáljuk az adatokat és az azt kezelõ mûveleteket. A régi input-output függvényeket helyettesíthetjük a C++ új iostream.h definícióival. A vermet kezelõ elsõ programunkat így a következõképpen írhatjuk át az objektum-orientált szemléletre.

```
#include <iostream.h>

    const  stLEN = 20;

struct Stack {
    double s[ stLEN ];
    int    sp;

    Stack() { sp = -1; }
    void Push( double v ) { s[ ++sp ] = v; }
    double pop() { return s[ sp-- ]; }
   };


void main() {
    Stack s;

    s.Push( 1 ); s.Push( 2 ); s.Push( 3 );
    cout << s.pop() << ", ";
    cout << s.pop() << ", ";
    cout << s.pop() << '\n';
   }
```

Az átírt program valószínûleg egyetlen gépikódú utasításában sem fog eltérni a C nyelven megírt megfelelõjétõl. A szemlélet jelölése "mindössze" azt teszi lehetõvé, hogy a szorosabban összetartozó adatokat és az azt kezelõ mûveleteket egyetlen egységként adjuk meg. Az eszközökkel a programban "összevissza" elhelyezkedõ, de logikailag szorosan egymáshoz kapcsolódó elemeket egyetlen helyre tudtuk gyûjteni. Az egységbezárás mellett a bezárást úgy valósítottuk meg, hogy közvetlenül nem hivatkoztunk az objektum megvalósítására, az egyes attribútumokra.

Az eszközkészlet kissé mélyebb ismeretével tovább egyszerûsíthetjük a verem-objektumra történõ hivatkozásokat. Így definiálhatunk egy olyan módszert (top), amellyel a verem legfelsõ elemét elérhetjük, sõt, azt módosíthatjuk is (például: s.top() = 5.3). Az objektumon egyszerû transzformációt végrehajtó, de értéket vissza nem utaló módszer esetén legyen a visszatérési érték mindig maga az objektum, ekkor ugyanis az objektumon végrehajtott ilyen jellegû transzformációk sorozatát csõvezetékbe (pipe) rendezhetjük (ez sajnos még nem tartozik az objektum-orientáltság szemléletébe). Ez tovább egyszerûsíti a hívást. Az objektum "végighalad" a (Unix pipe-jához hasonló) csõvezetéken, miközben azon a megadott transzformációk sorban végrehajtódnak. Végül magát a kiírómûveletet is definiálhatjuk a veremre, így az ismételt kiírások sorozatát is lerövidíthetjük.

```
#include <iostream.h>
```

```
    const  stLEN = 20;

struct Stack {
    double s[ stLEN ];
    int    sp;

    Stack() { sp = -1; }
    Stack& Push( double v ) { s[ ++sp ] = v; return *this; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
   };
ostream& operator <<( ostream& o, Stack& s ) { o << s.pop(); return o; }

void main() {
    Stack s;

    s.Push( 1 ).Push( 2 ).Push( 3 );
    cout << s << ", " << s << ", " << s << '\n';
   }
```

A definiált veremből származtathatunk egy olyan osztályt, amely alkalmas a verem segítségével a fordított lengyel jelölés (RPN) kiértékelésére. A példában egy kiegészítõ mûvelet a verem tetején elhelyezkedõ két elemen végrehajt egy kétoperandusú mûveletet. Sajnos a csõvezeték, pontosabban az egy adott objektumon végrehajtott transzformációk csõvezeték-jellegû megadása még nem koncepcionális eleme a C++-nak, így a jelölést csak közvetett módon tudjuk fenntartani. A transzformációs mûveleteket meg kell ismételnünk az RPN osztály definiálásakor: a mûveleteket egyszerûen delegáljuk az õsosztálynak. (A megoldás az õsosztály definíciójában egy olyan típus megadásának lehetõsége lenne, amely minden leszármazott osztály esetén magának a leszármazott osztálynak a típusával egyezik meg: ekkor szükségtelen lenne a módszerek újradefiniálása.)

```
#include <iostream.h>

    const  stLEN = 20;

struct Stack {
    double s[ stLEN ];
    int    sp;

    Stack() { sp=-1; }
    Stack& Push( double v ) { s[ ++sp ] = v; return *this; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
   };

struct RPN : Stack {
    RPN() :Stack() {}

    RPN& Push( double v ) { Stack::Push( v ); return *this; }
    RPN& Op( char c ) { double y = pop(),  x = pop();
        switch( c ) {
          case '+': x += y; break;
          case '-': x -= y; break;
          case '*': x *= y; break;
          case '/': x /= y; break;
         }
        Push( x );
        return *this; }
   };

ostream& operator <<( ostream& o, Stack& s ) { o << s.pop(); return o; }

void main() {
    RPN s;
```

```
    s.Push( 1 ).Push( 2 ).Push( 3 ).Op( '*' ).Op( '+' ); cout << s << '\n';
   }
```

Mivel a verem ideiglenes változóját csak a transzformációsorozat és a kiírás esetén használtuk fel, a hívás (a fõprogram) így tovább egyszerûsíthetõ:

```
void main() {

    cout << RPN().Push(1).Push(2).Push(3).Op('*').Op('+') << '\n';
   }
```

Természetesen elképzelhetõ, hogy mélyen egymásbaágyazott kifejezések esetén nem lesz elegendõ a konstansként megadott tömbméret. Mivel a bezárás technikáját alkalmaztuk, ezért a verem megvalósítását megváltoztathatjuk anélkül, hogy az kihatna a program más részeire. A változtatás egyik lehetséges módja, hogy a tömböt a dinamikus tárterületre helyezzük és így méretét az objektum készítésekor is meghatározhatjuk. Az alapértelmezett-operandusérték definiálásának lehetõségével elérhetjük azt, hogy a program a korábbi változatokkal megegyezõ módon mûködjön, ha a hosszat külön nem adjuk meg. Csak a konstruktort változtattuk, valamint az objektumot kiegészítettük egy destruktorral is. A C mutató-aritmetikájának köszönhetõen az adatokat felhasználó többi módszeren sem kell változtatnunk.

```
struct Stack {
    int     len;
    double *s;
    int     sp;

    Stack( int l = stLEN ) { sp = -1; s = new double [ len = l ]; }
    ~Stack() { delete s; }
    Stack& Push( double v ) { chk(); s[ ++sp ] = v; return *this; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
   };
```

További változtatásokkal elérhetjük, hogy a verem mérete automatikusan a szükséges mértékben növekedjen. (Ha az stLEN értékét 1-re módosítjuk, akkor nyomkövetéssel követhetjük a verem változását.)

```
struct Stack {
    int     len;
    double *s;
    int     sp;

    Stack& chk() { if( len <= sp+1 ) {
                      double *z = s; int l = len; double v;
                      s = new double [ len = sp+stLEN+1 ];
                      for( ; --l >= 0; s[l] = z[l] ) ;
                     }
                  return *this;
                 }
    Stack( int l = stLEN ) { sp = -1; s = new double [ len = l ]; }
    Stack& Push( double v ) { chk(); s[ ++sp ] = v; return *this; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
   };
```

Mivel a C++ rendelkezik a bezárás eszközkészletével is, ezért bejelölhetjük az objektum implementációs részét és az interfészt, a publikus felületet. A jelölés mellett a C++ nyelvi mechanizmusa egyben azt is biztosítja, hogy a program más részeibõl nem érhetõk el az objektum "saját", implementáció-függõ részei.

```
class Stack {
    int     len;
    double *s;
    int     sp;
```

```
    Stack& chk() { if( len <= sp+1 ) {
                       double *z = s; int l = len; double v;
                       s = new double [ len = sp+stLEN+1 ];
                       for( ; --l >= 0; s[l] = z[l] ) ;
                      }
                   return *this;
                  }
   public:
    Stack( int l = stLEN ) { sp = -1; s = new double [ len = l ]; }
    Stack& Push( double v ) { chk(); s[ ++sp ] = v; return *this; }
    double& top() { return s[ sp ]; }
    double  pop() { return s[ sp-- ]; }
   };
```

A verem implementációját ugyanilyen könnyen átalakíthatjuk láncolt listává.

```
#include <iostream.h>

    const NIL = 0L;

class Stack {
    struct elem {
        elem   *N;
        double  v;

        elem(elem *oN, double ov) :N(oN), v(ov) { }
       };

    elem   *s;

   public:
    Stack() { s = NIL; }
    Stack& Push( double v ) { s = new elem(s,v); return *this; }
    double& top() { return s->v; }
    double  pop() { double r = s->v; elem *z;
                    s = (z = s)->N, delete z; return r; }
   };

struct RPN : Stack {
    RPN() :Stack() { }

    RPN& Push( double v ) { Stack::Push( v ); return *this; }
    RPN& Op( char c ) { double y = pop(),  x = pop();
        switch( c ) {
          case '+': x += y; break;
          case '-': x -= y; break;
          case '*': x *= y; break;
          case '/': x /= y; break;
         }
        Push( x );
        return *this;
       }
   };

ostream& operator <<( ostream& o, Stack& s ) { o << s.pop(); return o; }

void main() {

    cout << RPN().Push(1).Push(2).Push(3).Op('*').Op('+') << '\n';
   }
```

A C++ eszközkészletének segítségével a verem implementációjának változtatásait úgy tudtuk végrehajtani, hogy közben nem kellett módosítanunk az azt felhasználó programrészleteken a fõprogramban, sõt a verembõl leszármaztatott osztály esetén sem. A bezárás technikájának további felhasználásával a program

úgy módosítható, hogy az egyes implementációs változatok a rendszerben egymás mellett is létezhessenek. Ehhez mindössze egy egyszerû változtatásra, az RPN osztály template-osztályként történõ definiálására (hívása például: RPN<listStack>) van szükség.

Irodalomjegyzék

- M. Atkinson, D. DeWitt, D. Maier, F. Bancilhon, K. Dittrich, S. Zdonik, "The Object Oriented Database System Manifesto"
- Bjarne Stroustrup. The C++ programming language. (2nd ed.) Addison-Wesley, 1994.
- J. Rumbaugh et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
- J. Rumbaugh et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.
- J. Rumbaugh et al. Object-Oriented Modeling and Design. Prentice-Hall, 1991.