Utilizarea şi supraîncărcarea operatorilor NEW şi DELETE

Limbajul C++ a extins modelul gestionării memoriei din limbajul C prin introducerea construcțiile necesare pentru a suporta obiecte. Pentru a construi variabile dinamice C-ul foloseşte funcții de librărie (*malloc()*/*free()*); în schimb, C++ introduce operatorii *new*/*delete*. Funcțiile C stau în continuare la dispoziția programatorului C++; compatibilitatea cu C este utilă, în special, pentru a include cod C în programele C++ şi pentru a scrie cod C++ care să poată fi suportat de medii C, dar şi pentru a face posibilă implementarea operatorilor *new*/*delete* prin funcțiile de librărie.

I. Utilizarea operatorilor NEW şi DELETE

I.1 malloc()/free() vs. new/delete

Ce va alege un programator C++ pentru a gestiona memoria dinamică? Atunci când el nu este constrâns, din diverse motive, să apeleze la *malloc()/free()*, va alege întotdeauna *new/delete*. De ce? Pentru că *malloc()/free()* nu suportă semnatica obiectelor, nu sunt extensibile, iar *new/delete* sunt semnificativ mai siguri!

malloc()/*free*() nu fac altceva decât să aloce/dealoce zone de memorie; ele nu sunt capabile să construiască/distrugă obiecte în/din memoria alocată/dealocată (adică nu pot apela constructorul/destructorul)! Încercarea de a crea un obiect cu *malloc*() conduce la comportament nedefinit (iar "nedefinit" însemnă, de cele mai multe ori, că programul vostru va funcționa în timpul dezvoltării, va funcționa în teste şi va genera erori în timpul prezentării sale celui mai important client!):

```
//exemplul 1
string* pstr = static_cast<string*> (malloc (sizeof (string)));
//orice încercare de a utiliza pstr ca pe un pointer la string produce un comportament nedefinit!
std::string* stringArray1 = static_cast<std::string*> (malloc(10 * sizeof (std::string)));
free (stringArray1);
```

stringArray1 referă suficientă memorie pentru 10 obiecte *std::string*, dar aceste obiecte nu sunt inițializate; chiar presupunându-le inițializate, *free*() nu va apela destructorii acestor obiecte, ceea ce înseamnă că va fi pierdută toată memoria pe care aceste obiecte şi-au autoalocat-o intern pentru a păstra şiruri de caractere!

Dacă *malloc*() necesită ca programatorul să specifice numărul de octeți necesari, *new* calculează singur dimensiunea obiectelor pe care le construieşte; în timp ce *malloc*() returnează *void**, fiind necesară o conversie, *new* returnează direct un pointer la tipul solicitat:

```
//exemplul 2
int* p = static_cast<int*> malloc(sizeof(int));
int* q = new int;
```

Operatorii *new* şi *delete* pot fi supraîncărcați pentru o clasă, ceea ce permite implementarea unei politici specifice clasei pentru gestionarea memoriei, aşa cum vom vedea într-un viitor exemplu; *malloc*() nu poate fi supraîncărcată.

Mixarea lui *new*/*delete* cu *malloc*()/*free*() este, în general, o idee rea; apelarea lui *free*() pentru un pointer obținut cu *new* şi apelarea lui *delete* pentru un pointer obținut cu *malloc*() generează comportament nedefinit! Nu putem presupune că implementarea lui *new*/*delete* utilizează *malloc*()/*free*(); standardul C++ interzice în mod explicit doar implemetarea lui *malloc*()/*free*() prin *new*/*delete*. Însă standardul C++ face o diferențiere clară între zonele de memorie dinamică accesate prin funcțiile din librăria C (zonă numită *heap*) şi prin operatorii C++ (zonă numită *free store*). În afară de a fi accesate diferit, heap-ul şi free store au, conform standardului C++, proprietăți diferite, ceea ce sugerează implementări independente!

I.2 Câte forme de new/delete există?

Dacă operatorul *new* (plain *new*) alocă un obiect (un apel de constructor), operatorul *new*[] (array *new*) alocă un tablou de obiecte (câte un apel de constructor pentru fiecare componentă a tabloului). Evident, există şi perechile acestor operatori, *delete* (plain *delete* – un apel de destructor) şi *delete*[] (array *delete* – câte un apel de destructor pentru fiecare componentă a tabloului). În C++ pre-standard, dacă

încercarea de a aloca memorie eşua, *new* şi *new*[] returnau NULL; standardul C++ stipulează că cei doi operatori vor genera, într-o astfel de situație, o excepție de tip std::bad_alloc!

```
//exemplul 3
string *p = new string[10];
if (p == NULL){
        //OK, până în 1994
        //p se poate utiliza
        ...
        delete [] p;
}
```

Politica de a returna NULL forțează programatorul să testeze de fiecare dată valoarea returnată, ceea ce poate genera foarte uşor erori; de asemenea, aceste teste măresc dimensiunea programului şi induc suprasarcini la execuție. Eşecul alocării de memorie dinamică este o situație ce se produce rar şi indică, cel mai probabil, că sistemul se află într-o stare instabilă; adică exact acel tip de erori de execuție pe care excepțiile au fost proiectate să le trateze!

```
//exemplul 4
#include <iostream>
#include <stdexcept>

const long int BUF_SIZE = 99999999L;

int main(){
        try{
                double *pd = new double [BUF_SIZE];
                //utilizare sigură a lui pd
                delete [] pd;
        }
        catch(std::bad_alloc& e){
                std::cout << e.what() << std::endl;
                //tratarea exceptiei
        }
        return 0;
}
```

Un program care apelează *new*, direct sau indirect (de exemplu, containerele STL îşi alocă singure, în mod automat, memorie din free store), trebuie deci să conțină un handler pentru tratarea excepției std::bad_alloc. De asemenea, testarea pointerului returnat de *new* devine inutilă. Aceasta deoarece, dacă apelul reuseşte, testul se efectuează degeaba, iar dacă apelul nu reuşeşte testul nu mai este executat deloc, firul curent de execuție a programului fiind întrerupt!

Câteodată însă nu este de dorit a se genera excepții; de exemplu, excepțiile nu sunt încă suportate de implementare sau au fost oprite pentru a mări performanța (se ştie că excepțiile sunt tratate la execuție, cu costurile aferente). Din acest motiv, standardul C++ include şi versiunile lui *new*/*new*[] care nu generează excepții (ci returnează NULL). Aceste două versiuni necesită un argument suplimentar, de tip const std::nothrow_t&, tip definit în headerul <new>.

```
//exemplul 5
#include <iostream>
#include <new>
#include <string>

const long int BUF_SIZE = 99999999L;

int main(){
        double *pd = new (std::nothrow)  double[BUF_SIZE];
        if (pd == NULL)
                std::cout << "NEW[] a esuat" << std::endl;
        delete [] pd;

        std::string *ps = new (std::nothrow) std::string;
        if (ps == NULL)
                std::cout << "NEW a esuat" << std::endl;
        delete ps;
        return 0;
}
```

Există deci nothrow *new* şi nothrow *new*[], versiuni care nu generează excepții. Headerul <new> conține definiția de clasă:

struct nothrow_t{};

(adică o clasă vidă, al cărei singur scop este acela de a ajuta la supraîncărcarea lui *new*); argumentul de apel al operatorului este definit prin:

extern const nothrow_t nothrow;

Dar asta nu e totul! Se poate construi un obiect (sau un tablou de obiecte) şi la o adresă de memorie determinată anterior; aceste versiuni sunt numite placement *new* şi placement *new*[] şi au o mulțime de aplicații (mai ales dacă ținem seamă de faptul că memoria fiind alocată anterior, aceste versiuni sunt mai rapide iar eşecul la alocare nu poate apărea!). Placement *new*/*new*[] acceptă, suplimentar, un argument de tip void* care referă zona de memorie în care se va construi obiectul/taboul de obiecte.

```
//exemplul 6
#include <iostream>
#include <new>
class C{
        int a;
        public:
        C(int aa): a(aa)    { std::cout << "ctor" << std::endl; }
        ~C()                { std::cout << "dtor" << std::endl; }
        int f()             { std::cout << a << std::endl; }
};

//alocare buffer
char tc [sizeof(C)];

int main(){
        //apel placement new
        C* pC = new (tc) C(10);
        //utilizare pC
        pC->f();
        //apel explicit pentru destructor!!!
        pC->C::~C();
        return 0;
}
```

Destructorii obiectelor construite cu placement *new*/placement *new*[] trebuie apelați explicit (de ce? citeşte II.2); altfel, ei nu vor fi apelați deloc, ceea ce însemnă că obiectele nu vor elibera corect resursele achiziționate la construcție sau pe parcursul vieții lor.

I.3 Utilizați întotdeauna new şi delete în forme corespondente!

Ce este în neregulă cu următorul fragment de cod?

```
//exemplul 7
string *ps = new string [10];
delete ps;
```

Acest fragment de cod va genera comportament nedefinit. De ce? Pentru că alocarea de memorie se realizează cu *new*[], ceea ce are ca efect inițializarea a 10 obiect std::string (adică 10 apeluri de constructor), în timp ce dealocarea de realizează cu *delete*, care apelează doar destructorul primului obiect din tablou. Practic, din moment ce atât *new* cât şi *new*[] returnează doar un pointer, *delete* trebuie să se întrebe: “Pointerul primit ca argument referă un obiect sau un tablou de obiecte?” Iar *delete* nu poate răspunde la această întrebare decât dacă programatorul îi oferă răspunsul prin folosirea sau nu a operatorului de indexare []! Apelarea lui *delete*

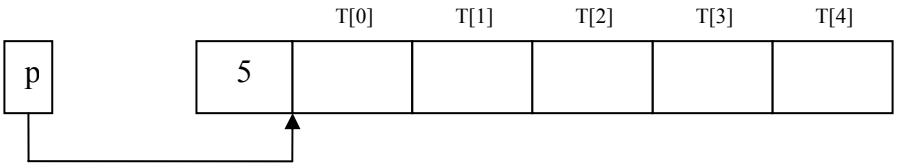
pentru a elibera o zonă de memorie alocată cu *new*[] generează comportament nedefinit; analog, apelarea lui *delete*[] pentru a elibera o zonă de memorie alocată cu *new* generează comportament nedefinit. Ca o consecință imediată, evitați introducerea de aliasuri către tipuri tablou prin intermediul lui typedef deoarece pot cauza confuzie:

```
//exemplul 8
typedef std::string message[4];
std::string *ps = new message;      //plain new
delete ps ;                         //plain delete - comportament nedefinit!
delete [] ps;                       //array delete - corect!
```

Cum ştie delete[] câte obiecte are de distrus?

```
//exemplul 9
T* p = new T[5];
delete [] p;
```

Această informație este păstrată într-o manieră dependentă de implementare; în general, *new*[] alocă suplimentar o zonă de memorie de dimensiune sizeof(size_t) octeți, zonă în care scrie numărul obiectelor alocate:



Când este invocat *delete*[], acesta cercetează această zonă de memorie pentru a afla dimensiunea tabloului; el invocă apoi de 5 ori destructorul clasei T şi în final eliberează memoria. *delete* nu face aceste operații, de aici rezultând explicația afirmațiilor anterioare.

II. Supraîncărcarea operatorilor NEW şi DELETE

II.1 Cum funcționează new şi delete?

Operatorii *new*/*delete* sunt invocați într-un program prin *expresii new*/*delete*. O *expresie new* apelează repetat o *funcție de alocare* a memoriei; dacă alocarea nu reuşeşte, se generează o excepție de tip std::bad_alloc; dacă alocarea reuşeşte, un

obiect este construit (prin apelarea unui constructor) în zonă de memorie alocată. O *expresie delete* apelează destructorul, urmat de o *funcţie de dealocare* a memoriei.

Acest comportament este "built into the language" nu poate fi modificat (tot aşa cum nu poate fi modificat comportamentul lui sizeof)!!!

II.2 De fapt, pe cine supraîncărcăm?

Aşa cum spuneam, comportamentul *operatorilor new/delete* nu poate fi modificat. Poate fi modificat, în schimb, felul în care aceşti operatori alocă/dealocă memorie prin supraîncărcarea *funcţiilor de alocare şi dealocare*; expresia new/delete va selecta funcţia de alocare/dealocare corespunzătoare.

C++ defineşte următoarele funcţii globale de alocare şi dealocare a memoriei:

```
//exemplul 10
// extrase din <new>
void* operator new(std::size_t)      throw (std::bad_alloc);
void* operator new[](std::size_t)    throw (std::bad_alloc);
void operator delete(void*)          throw();
void operator delete[](void*)        throw();
void* operator new(std::size_t, const std::nothrow_t&)        throw();
void* operator new[](std::size_t, const std::nothrow_t&)      throw();
void operator delete(void*, const std::nothrow_t&)            throw();
void operator delete[](void*, const std::nothrow_t&)          throw();
//placement new şi placement delete
inline void* operator new(std::size_t, void* __p)             throw() { return __p; }
inline void* operator new[](std::size_t, void* __p)           throw() { return __p; }
inline void  operator delete  (void*, void*) throw() { };
inline void  operator delete[](void*, void*) throw() { };
```

Observaţi că *operatorul new* alocă memorie apelând funcţia de alocare numită operator new()! Parametrul de tip size_t specifică câtă memorie trebuie alocată. Tipul returnat este void* pentru că funcţia de alocare nu face altceva decât să rezerve o zonă de memorie din free store (aşa cum *malloc*() rezervă memorie din heap); puteţi apela explicit o funcţie de alocare a memoriei:

```
//exemplul 11
//nu va fi apelat constructorul clasei string!
void *rawMemory = operator new (sizeof(std::string));
```

Este sarcina *operatorului new* să preia această zonă de memorie şi să o transforme într-un obiect prin apelarea unui constructor. O *expresie new* precum:

```
std::string *ps = new std::string("pe cine supraincarcam?");
```

este tradusă de compilator în (aproximativ) următorul mod:

```
void *rawMemory = operator new (sizeof(std::string));          //alocare memorie din free store
std::string::string(rawMemory, “pe cine supraincarcam?”);      //construcție obiect
std::string *ps = static_cast<string*>(rawMemory);             //returnare pointer convertit
```

Operația inversă o realizează *operatorul delete*; acesta apelează mai intâi destructorul, iar apoi funcția de dealocare.

Funcțiile globale de alocare/dealocare corespunzătoare lui placement new/delete nu pot fi supraîncărcate/redefinite. Se observă din exemplul 10 că funcția de dealocare corespunzătoare lui placement delete nu face nimic; este evident acum, la exemplul 6, de ce destructorii obiectelor construite cu placement new trebuie apelați explicit.

Celelalte funcții globale de alocare pot fi supraîncărcate/redefinite, dar nu este recomandabil (redefinirea funcției globale de alocare *operator new* poate genera incompatibilități cu o librărie care face acelaşi lucru!); se recomandă, în schimb, supraîncărcarea acestor funcții în clase. Dacă funcțiile globale de alocare sunt redefinite (acelaşi prototip) sau supraîncărcate (prin adăugarea de parametri suplimentari în prototip), întotdeauna primul argument va fi de tipul size_t.

II.3 De ce să supraîncărcăm funcțiile de alocare?

În general, răspunsul la această întrebare este unul singur: eficiența! Fiind adaptate perfect utilizării la modul general, versiunile implicite ale lui *operator new* şi *operator delete* sunt inflexibile în anumite situații concrete, conducând la penalități referitoare la viteza de execuție; un exemplu clasic sunt aplicațiile care alocă dinamic un număr foarte mare de obiecte de mici dimensiuni.

Deoarece versiunea implicită a lui *operator new* trebuie să fie “bună la toate”, ea trebuie să fie pregătită să aloce blocuri de memorie de orice dimesiune; analog, versiunea implicită a lui *operator delete* trebuie să fie pregătită să dealoce blocurile de memorie pe care *operator new* le-a alocat. Dar ce dimensiune au aceste blocuri? Reamintim că *operator delete* este apelat după destructor şi primeşte ca argument un void*! În mod clar, *operator new* trebuie să-i comunice lui *operator delete*

dimensiunea blocului de memorie alocat; această cerință se rezolvă, de obicei, într-o manieră asemănătoare cu cea prezentată la I.3, prin alocarea pentru fiecare bloc a unei zone suplimentare în care se vor completa informațiile necesare dealocării blocului. Pentru obiecte de dimesiuni mici (de exemplu, obiecte care conțin doar un pointer[1]) rezultă imediat că memoria alocată poate fi chiar dublul celei efectiv necesare; într-un mediu în care memoria disponibilă este redusă acest fapt poate fi inacceptabil.

II.3 Supraîncărcarea funcțiilor de alocare în clase

Supraîncărcarea funcțiilor se alocare în clase trebuie realizată într-o manieră consistentă cu comportamentul funcției de alocare globală. Aceasta însemnă, în cazul unei alocări reuşite, returnarea unui pointer către zona alocată; în cazul unei alocări nereuşite, *operator new* apelează, prin intermediul unui pointer la funcție, o rutină numită new-handler. De fapt, se intră într-o buclă repetitivă, în care, după fiecare eşec de alocare, *operator new* apelează *new-handler*, ipoteza fiind că această funcție reuşeşte, cumva, să facă rost de mai multă memorie. Din această buclă repetitivă se poate ieşi printr-o alocare reuşită sau prin setarea pe NULL a pointerului către *new-handler*, caz în care funcția de alocare va genera o excepție std::bad_alloc. Următorul algoritm în pseudocod descrie acest comportament:

```
void* operator new (size_t dimensiune){
        if (dimensiune == 0)   dimensiune = 1;
        while (true){
                încearcă să aloci dimensiune octeți;
                if (alocarea a avut succes) return (pointer la zona alocată);

                //alocarea nu a reuşit; salvează new-handler-ul curent
                new_handler actual_handler = set_new_handler(0);
                //reinstalează new-handler-ul curent
                set_new_handler(actual_handler);
                //este new-handler-ul curent diferit de NULL?
                if (actual_handler)
                        (*actual_handler)();
                else
                        throw std::bad_alloc();
                }
        }
```

[1] Vezi "*Item 30 – The Fast Pimpl Idiom*" în "*Exceptional C++*" de Herb Sutter, Addison Wesley, 1999

Standardul C++ cere operatorului *new* să returneze un pointer valid chiar şi în cazul unei cereri de alocare de 0 octeți; un truc simplu şi legal este ca, într-o astfel de situație, să alocăm un octet.

Dacă încercarea de alocare nu reuşeşte, funcția de alocare apelează repetat rutina *new-handler*; header-ul <new> conține următoarele declarații:

```
//exemplul 12
// extras din <new>
/** If you write your own error handler to be called by @c new, it must be of this type. */
typedef void (*new_handler)();
/// Takes a replacement handler as the argument, returns the previous handler.
new_handler   set_new_handler(new_handler) throw();
```

Cu alte cuvinte, *new_handler* este un alias pentru un pointer la funcție care nu returnează şi nu ia argumente; *set_new_handler*() este o funcție care instalează new_handler-ul primit ca argument şi îl returnează pe cel vechi. Ce poate face un new_handler pentru a obține mai multă memorie? O primă tehnică ar consta în alocarea, la începutul programului, a unui bloc mare de memorie; prima dată când new_handler-ul ar fi apelat, el ar elibera acest bloc şi, eventual, ar avişa un mesaj de atenționare. O altă tehnică ar putea consta în instalarea, cu *set_new_handler*(), a unui nou *new_handler*, care s-ar putea dovedi mai eficient. Alte variante posibile ar fi apelarea lui abort(), generarea unei excepții std::bad_alloc() sau de-instalarea new-handler-ului, ceea ce va conduce tot la generarea excepției std::bad_alloc().

Pentru a şti dacă este instalat sau nu un *new_handler*, s-a recurs la următoarea tehnică: deinstalarea şi salvarea *new_handler*-ului curent într-o variabilă auxiliară, urmată de reinstalarea sa şi testarea față de 0 a variabilei auxiliare.

Prezentăm în continuare un exemplu de utilizare a lui *set_new_handler*():

```
//exemplul 13
#include <iostream>
#include <stdexcept>
#include <new>
#include <stdlib.h>
const long int BUF_SIZE = 99999999L;

//new_handler
void noMoreMemory(){
        static unsigned int i=0;
        ++i;
        std::cout << "nu pot satisface aceasta cerere de memorie!\t" << i << std::endl;
        if (i>=3){ std::set_new_handler(0); i=0;
        }
}
```

```
class C{
	double d;
	public:
	static void* operator new[](size_t);
};

void* C::operator new[](size_t d){
	std::cout << "personal operator new[]!" << std::endl;
	//apelăm versiunea globală
	::operator new[](d);
}

int main(){
	std::set_new_handler(noMoreMemory);
	try{
		*pC = new C[BUF_SIZE];
	}
	catch (std::bad_alloc& e){
		std::cout << e.what() << std::endl;
	}
	system("PAUSE");

	std::set_new_handler(noMoreMemory);
	try{
		double *pC = new double[BUF_SIZE];
	}
	catch (std::bad_alloc& e){
		std::cout << e.what() << std::endl;
	}
	system("PAUSE");
	return 0;
}
```

Clasa C redefineşte funcția de alocare globală *operator new*[], dar nu face altceva decât să anunțe acest lucru prin afişarea unui mesaj şi să apeleze versiunea globală (observați sintaxa *::operator new*[]!). Se remarcă faptul că redefinirea se realizează printr-o funcție statică; chiar în lipsa lui static, această funcție ar fi tot statică. Se recomandă însă folosirea lui static, pentru a evidenția suplimentar această calitate. Execuția programului ne arată că, după 3 încercări nereuşite de a aloca memorie, este generată o excepție std::bad_alloc (deoarece, după 3 apeluri ale lui noMoreMemory(), aceasta instalează handlerul NULL). Se remarcă apoi că *new_handler*-ul instalat este la nivel global şi nu la nivel de clasă!

C++ nu suportă direct un *new_handler* specific unei clase; această funcționalitate poate fi însă fi simulată precum în exemplul următor:

```
//exemplul 14
class C{
	public:
	static void* operator new(size_t);
	static new_handler set_new_handler(new_handler);   //instalează un nou new_handler, specific clasei C
```

```
        private:
        static new_handler handlerActual;                    //new_handler-ul clasei C
};

new_handler C::handlerActual;                                //definiție; inițializare cu 0

new_handler C::set_new_handler(new_handler p){
        new_handler old = handlerActual;
        handlerActual = p;
        return old;
}

void* C::operator new(size_t d){
        //salvăm new_handler-ul global şi îl instalăm pe cel al clasei C
        new_handler global = std::set_new_handler(handlerActual);
        void *rawMemory;
        try{
                rawMemory = ::operator new(d);
        }
        catch (std::bad_alloc&){
                //restaurăm new_handler-ul global
                std::set_new_handler(global);
                //propagăm excepția
                throw;
        }
        //restaurăm new_handler-ul global şi în caz de reuşită
        std::set_new_handler(global);
        return rawMemory;
}

//utilizarea clasei C
void noMoreMemory();
//instalare new_handler specific clasei C
C::set_new_handler(noMoreMemory);
C *pC1 = new C;                         //dacă alocarea eşuează, apelează noNoreMemory()
string *ps = new string;                // dacă alocarea eşuează, apelează new_handler-ul global (dacă există!)
C::set_new_handler(0);                  //clasa C nu mai posedă new_handler
C *pC2 = new C;                         // dacă alocarea eşuează, generează imediat o excepție std::bad_alloc
```

Funcția globală de alocare *operator new* poate fi şi supraîncărcată în interiorul unei clase; în acest caz, funcția de alocare din clasă posedă parametri suplimentari (întotdeauna primul parametru este de tip size_t). Într-un astfel de caz trebuie să aveți grijă pentru că funcția de alocare din clasă o ascunde pe cea globală, aşa cum se observă în următorul exemplu:

```
//exemplul 15
class C{
        public:
        static void* operator new(size_t, new_handler);        //permite specificarea unui new_handler
};

void specialErrorHandler();

C *pC1 = new(specialErrorHandler) C;        //apelează C::operator new
C *pC2 = new C;                             //eroare!
```

Această situație se remediază dotând clasa C şi cu o versiune a lui *operator new* care să suporte invocarea versiunii globale:

```
//exemplul 16
class C{
        public:
        static void* operator new(size_t, new_handler);        //permite specificarea unui new_handler
        static voi* operator new(size_t d){
                return ::operator new(d);
        }
};
```

Un alt subiect care trebuie discutat este acela că funcția de alocare *operator new* este moştenită de subclase! Iar acest lucru este important pentru că o funcție de alocare specifică unei clase este proiectată şi optimizată pentru acea clasă, şi nu pentru subclasele sale (ale căror instanțe, foarte probabil, vor avea dimensiuni diferite)! Moştenirea face însă posibilă apelarea lui *operator new* pentru subclase:

```
//exemplul 17
class Base{
        public:
        static void* operator new(size_t);
        static void operator delete(void* rawMemory, size_t);
};
class Derived: public Base{
        int a;
};
Derived *pD = new Derived;        //apelează Base::operator new!!!
```

Din acest motiv trebuie să aveți grijă să protejați funcția de alocare specifică clasei, precum în următorul exemplu (evident, de un tratament asemănător trebuie să se bucure şi *operator delete*):

```
//exemplul 18
void* Base::operator new(size_t d){
        if (d != sizeof(Base))                    //dacă dimensiunea este incorectă
                return ::operator new(size);      //se apelează funcția globală de alocare
        ...
}
void* Base::operator delete(void* rawMemory, size_t d){
        if (rawMemory == 0) return;               //aplicarea lui delete peste NULL este o operație sigură
        if (d != sizeof(Base)){
                ::operator delete(rawMemory);
                return;
        }
        dealocă memoria referită de rawMemory;
        return;
}
```

II.4 Un exemplu de memory pool

```
//exemplul 19
class CalculatorImpl{
        //implementarea unei clase reprezentând un calculator
};

class Calculator{
        public:
        static void* operator new (size_t d);
        static void operator delete(void* rawMemory, size_t d);
        private:
        union{
                CalculatorImpl *pimpl;          //obiecte în uz
                Calculator      *next;          //memoria disponibilă
        };
        static const int BLOCK_SIZE;
        static Calculator *head;                //începutul memoriei disponibile
};

Calculator* Calculator::head = 0;
const int Calculator::BLOCK_SIZE = 512;

void* Calculator::operator new(size_t d){
        id (d != sizeof(Calculator))
                return ::operator new(size);
        Calculator *p = head;                   //începutul memoriei disponibile
        if (p)                                  //există memorie disponibilă?
                head = p->next;
        else {
                //nu există memorie disponibilă; alocăm suficientă memorie pt. BLOCK_SIZE obiecte
                Calculator *block = static_cast<Calculator*>(::operator new(BLOCK_SIZE * sizeof(Calculator)));
                //creăm o listă în blocul alocat
                for(int i=1; i<BLOCK_SIZE-1; ++i)
                        block[i].next = &block[i+1];
                block[BLOCK_SIZE-1].next = 0;
                p = block;                      //memoria returnată
                head = &block[1];               //memoria disponibilă
        }
        return p;
}

void Calculator::operator delete(void* rawMemory, size_t d){
        if (rawMemory == 0) return;
        if (d != sizeof(Calculator)){
                ::operator delete(rawMemory);
                return;
        }
        Calculator *carcasa = static_cast<Calculator*> (rawMemory);
        carcasa->next = head;
        head = carcasa;
}
```

Acest exemplu prezintă un concept care în literatura de specialitate este numit memory pool. Atunci când un obiect este distrus, memoria ocupată de el nu este returnată ci este păstrată în rezervă; la o alocare ulterioară, funcția *operator new* își preia memoria necesară din această rezervă (operațiune rapidă) și nu dintr-o alocare

efectivă din free store (operațiune lentă). Atunci când rezerva se goleşte, *operator new* recurge la alocare efectivă din free store. Deoarece *operator delete* nu returnează memoria către free store, dimensiunea unui *memory pool* poate creşte până la maximul memoriei necesare utilizatorului; un memory leak poate creşte indefinit!

În final, să remarcăm că, datorită faptului că *C::operator new* recurge la funcția globală de alocare pentru a obține memorie, protocolul descris la începutul II.3 este aplicat şi aici, fiind încapsulat în *::operator new*!

Norocel PETRACHE

Bibliografie:

1. Bjarne Stroustrup, *The C++ Programming Language*
2. *ANSI/ISO C++ Professional Programmer's Handbook*
3. Scott Meyers, *Effective C++*
4. Scott Meyers, *More Effective C++*
5. Herb Sutter – *Exceptional C++*
6. Bruce Eckel – *Thinking in C++*