Capitolul 11

13 Tratarea excepțiilor

Există situații în care pot apare erori în funcționarea unui program, mai ales în cazurile de interacțiune a programului cu exteriorul, când anumite valori nu pot fi controlabile în program.

De exemplu, cele mai multe funcții standard de intrare/ieșire returnează o valoare ce indică dacă operația respectivă s-a terminat sau nu cu succes. În cazul în care operația a eșuat, rezultă o situație excepțională în program, care trebuie tratată, astfel încât programul să nu conducă la erori deosebite. Din acest motiv, este indicat ca după fiecare apel al unei funcții standard ce poate eventual eșua, să se testeze modul de terminare al operației respective:

```
ifstream fout("f_out.txt");
if (!fout)
{
  // Tratarea cazului de eroare de deschidere fisier
}
// Operatii efectuate in cazul in care nu sunt erori
```

Nu întotdeauna însă există informații suficiente pentru tratarea unei anumite erori în zona de program în care aceasta a fost detectată. În aceste cazuri, informațiile despre eroare trebuie raportate spre un context mai cuprinzător, în mod uzual în blocul funcției care apelează funcția unde este detectată eroarea.

Limbajul C permite trei moduri de tratare a acestor situații:

a) Returnarea informațiilor de eroare din funcția apelantă spre funcția apelată. Acestea se pot returna prin intermediul parametrilor, sau a valorii de return. În cazul în care nu se pot returna asemenea informații de eroare, se setează anumiți indicatori globali de eroare, care pot fi testați în funcția apelantă.

b) Utilizarea funcțiilor standard `raise` şi `signal`, pentru gestionarea semnalelor sistemului. Funcția `raise` are rolul de a genera un eveniment, iar `signal` de a determina ce acțiuni se execută când evenimentul respectiv apare.

c) Utilizarea funcțiilor standard `setjmp` şi `longjmp`, pentru a face un salt nelocal în cadrul programului. Aceste funcții permit saltul între diferite funcții ale programului: `setjmp` are rolul se a salva contextul programului, iar `longjmp` de a restaura contextul în caz de eroare.

Aceste trei moduri prezintă însă unele dezavantaje. Prima variantă, care este şi cea mai utilizată de către programatori, presupune scrierea unui cod destul de complicat pentru a trimite şi a selecta informațiile de eroare. Ultimele două variante sunt destul de dificil de utilizat şi presupun o foarte bună cunoaştere a bibliotecii standard a limbajului C.

Limbajul C++ oferă o altenativă la aceste trei variante, mai simplu de utilizat şi mai eficientă: noțiunea de ***excepție***, precum si un mecanism de ***tratare a excepțiilor***.

13.1 Generarea şi recepția excepțiilor

Excepțiile reprezintă o modalitate de control a execuției programelor în situații de eroare. Mecanismul de tratare a excepțiilor presupune următoarele acțiuni:

- Suspendarea execuției funcției curente care a detectat o eroare, generarea unei excepții şi a unui obiect asociat acesteia, care conține informații referitoare la eroare;
- Reluarea execuției programului în altă zonă, recepționarea excepției şi obiectului asociat şi efectuarea acțiunilor necesare pentru tratarea erorii.

Se observă faptul că tratarea excepțiilor presupune două elemente distincte:

- Transferul nelocal al execuției programului între două zone distincte ale programului, transfer realizat automat de către compilator;
- Generarea şi recepționarea unei excepții şi a unui obiect asociate acesteia între cele două zone ale programului.

Generarea unei excepții şi a unui obiect asociat se realizează prin intermediul unei instrucțiuni speciale, numită ***instrucțiunea `throw`***, care are sintaxa următoare:

```
throw [ <expresie> ] ;
```

De exemplu, presupunând că într-un program s-a definit o clasă *Err* care are ca dată membră un şir de caractere şi un constructor, următoarea instrucțiune generează o excepție:

```
throw Err("Eroare deschidere fisier");
```

Efectul execuției este următorul:

- se suspendă execuția funcției curente;
- se crează un obiect corespunzător valorii expresiei ce urmează cuvântului `throw`;
- acest obiect este returnat unui context mai cuprinzător, chiar dacă el nu corespunde tipului valorii de return a funcției.

Înainte de transferul execuției programului spre un alt context, se distrug toate obiectele din contextul curent, care au fost complet create până în momentul generării excepției. Obiectul aferent excepției va fi distrus ulterior, după recepționarea ei.

În cazul în care instrucțiunea `throw` apare în corpul unei funcții, mecanismul de tratare a excepțiilor termină forțat execuția funcției curente. În cazul în care nu se doreşte părăsirea funcției, atunci se poate utiliza un bloc special, numit blocul `try`.

Sintaxa unui bloc `try` este:

```
try {
  // Secventa de cod care poate genera exceptii
}
```

iar efectul lui este următorul: contextul spre care se trimite excepția este cel imediat exterior blocului `try`. În acest fel, recepționarea şi tratarea excepției se face în aceeaşi funcție din care s-a generat excepția.

Recepția unei excepții se realizează prin intermediul unei construcții speciale, numită ***clauză `catch`***, sau ***handler de tratare a excepției***. Sintaxa unui asemenea handler este:

```
catch (<tip> <ident>) {
  // Secventa de cod pentru tratarea exceptiei
  // asociata obiectului <ident>
}
```

unde, `<tip>` reprezintă tipul de date al obiectului asociat excepției, iar `<ident>` este privit ca un parametru formal şi reprezintă numele obiectului.

O clauză `catch` permite tratarea unei singure excepții şi trebuie să urmeze întotdeauna după un bloc `try`.

În cazul în care în blocul `try` sunt generate mai multe excepții de acelaşi tip, o singură clauză `catch` este suficientă pentru tratrea acestora. În cazul în care însă, în acest bloc sunt generate mai multe excepții de tipuri diferite, se pot specifica mai multe clauze `catch` consecutive, câte una pentru fiecare tip de excepție. De exemplu:

```
try {
  // secventa care poate genera doua tipuri de exceptii
} catch(type1 id1) {
  // tratarea exceptiei de tipul type1
} catch(type2 id2) {
  // tratarea exceptiei de tipul type2
}
```

Exemplu. Programul următor utilizează mecanismul de tratare a excepțiilor, pentru a simula generarea şi recepția unei excepții.

```
#include <iostream>
using namespace std;

class A {
public:
  A() { cout << “Constructor A\n”; }
  ~A() { cout << “Destructor A\n”; }
  void f();
};

class B {
public:
  B() { cout << “Constructor B\n”; }
  ~B() { cout << “Destructor B\n”; }
  void g();
};
```

```
void A::f() {
  cout << "f in clasa A\n";
  throw 1;
  cout << "Instructiune ce nu va fi executata\n";
}

void B::g() {
  A a;
  cout << "g in clasa B\n";
  a.f();
  cout << "Instructiune ce nu va fi executata\n";
}

void main() {
  B b;
  try {
    b.g();
  }
  catch (int k) {
    cout << "Tratarea exceptiei. k = " << k << endl;
  }
}
```

În programul precedent, în cadrul funcției *f* din clasa *A* se lansează o excepție. Execuția funcției *f* este întreruptă, dar excepția nu este recepționată în funcția apelantă *g*, ci doar în funcția `main`.

Observație. Un bloc try poate exista şi la nivelul unei funcții, caz în care înlocuieşte blocul acesteia. De exemplu:

```
void main() try {
  throw “Functia main”;
}
catch (const char* msg) {
  cout << msg << endl;
}
```

O secvență de clauze `catch` poate conține o ***clauză `catch` implicită***. Aceasta se specifică în mod uzual pe ultima poziție într-o asemenea secvență şi conține operatorul “...” în locul parametrului clauzei. O clauză `catch` implicită specifică faptul că handlerul respectiv tratează o excepție al cărei tip nu poate fi determinat (nu există un handler specific paentru tipul respectiv).

Exemplu.

```
#include <iostream>
using namespace std;

void main() {
```

```
  try {
    int n;
    cin >> n;
    switch (n) {
    case 1: throw 2;
    case 2: throw “Sir”;
    case 3: throw 12.25;
    }
  }
  catch (char const *mess) {
     cout << "Tratare char const * : " << mess << endl;
  }
  catch (int k) {
     cout << "Tratare int : " << k << endl;
  }
  catch (...) {
     cout << "Tip necunoscut\n";
  }
}
```

În concluzie, o excepție este generată de o instrucțiune `throw` şi este recepționată de prima clauză `catch`, exterioară contextului din care a fost generată excepția.

Limbajul C++ nu impune ca o anumită funcție care generează execpții să declare în mod explicit aceasta. Însă un stil bun de programare cere ca programatorul care defineşte o asemenea funcție să informeze potențialii utilizatori ai codului despre acest lucru.

Declararea execpțiilor generate de o funcție se poate face prin intermediul unei construcții specifice, numită ***specificarea excepțiilor***. Aceasta se inserează imediat după antetul funcției, conform sintaxei:

```
<antet functie> throw ‘(‘ <tip> {, <tip> } ‘)’
```

Observații.

1. Între paranteze se specifică tipurile de date ale obiectelor asociate excepțiilor generate de funcție. De exemplu, secvența:
   ```
   foid f() throw(int, A);
   ```
   specifică faptul că funcția *f* poate lansa excepții de două tipuri, asociate unor obiecte de tipul `int` şi *A*.

2. În cazul în care o funcție nu posedă după antet partea de specificarea a excepțiilor, aceasta înseamnă că ea poate lansa orice tip de excepții.

3. În cazul în care după cuvântul cheie `throw`, lista tipurilor este vidă, aceasta înseamnă că funcția respectivă nu generează excepții. Exemplu:
   ```
   foid f() throw();
   ```

13.2 Cazuri specifice de tratare a excepțiilor

Există anumite cazuri specifice ce pot apare la generarea şi recepția excepțiilor. În acest paragraf se vor descrie două situații: cazul excepțiilor netratate şi a celor retransmise spre alt context.

A. Excepții netratate

Există situații în care pentru o funcție care generează excepții se specifică o anumită listă de excepții, însă datorită unor anumite motive, este generată o excepție diferită de cele declarate. De exemplu, în funcția respectivă se poate apela o altă funcție, despre care nu se ştie dacă generează excepții şi eventual ce tipuri de excepții generează.

În această situație, în care se generează o excepție diferită de cele specificate şi nici una dintre clauzele `catch` nu o tratează, sistemul lansează în mod implicit în execuție o funcție predefinită `unexpected`, care la rândul ei apelează funcția sistem `terminate`.

Pentru a specifica o funcție proprie care să fie apelată în această situație, se poate utiliza o altă funcție predefinită, `set_unexpected`. Funcția `set_unexpected` primeşte ca parametru un pointer la o funcție de tipul `void()`, ce reprezintă funcția ce se va executa în această situație.

Există unele compilatoare care nu utilizează funcția `unexpected`. În această situație, se poate utiliza funcția predefinită `set_terminate`, pentru a specifica funcția proprie ce se va executa în această situație.

Exemplu. Programul următor utilizează funcția `set_terminate` pentru a specifica funcția care se va executa în cazul în care compilatorul nu găseşte nici un handler specific pentru o excepție.

```
#include <iostream>
using namespace std;

class A { };
class B { };

void g();

void f(int i) throw (A, B) {
  switch(i) {
    case 1: throw A();
    case 2: throw B();
  }
  g();
}

void g() { throw 47; }

void my_terminate() {
  cout << "Exceptie necunoscut\n";
  exit(1);
}
```

```
void main() {
  set_terminate(my_terminate);
  for(int i = 1; i <=3; i++)
    try {
      f(i);
    }
    catch(A) {
      cout << "Tratare exceptie A" << endl;
    }
    catch(B) {
      cout << "Tratare exceptie B" << endl;
    }
}
```

B. Retransmiterea excepțiilor

O altă situație specifică ce poate apare în cazul excepțiilor o constituie cea în care o anumită excepție este recepționată într-un context din program, în care nu există încă suficiente informații pentru tratarea ei. În acest caz, excepția este ***retransmisă*** spre un context mai cuprinzător din program, toate informațiile aferente excepției rămânând nemodificate

Retransmiterea unei excepții se face cu ajutorul instrucțiunii `throw` fără o expresie atașată:

```
throw;
```

Exemplu. Reluarea unui exemplu precedent, în care handlerul implicit retransmite excepția contextului imediat superior.

```
#include <iostream>
using namespace std;

void main() {
  try {   // Nivel 0
    try { // Nivel 1
      int n;
      cin >> n;
      switch (n) {
      case 1: throw 2;
      case 2: throw “Sir”;
      case 3: throw 12.25;
      }
    }
    catch (char const *mess) {
      cout << "Nivel 1. char const * : " << mess << endl;
    }
    catch (int k) {
      cout << "Nivel 1. int : " << k << endl;
    }
    catch (...) {
      cout << "Nivel 1. Tip necunoscut\n";
      throw;   // Retransmitere exceptie
    }
```

```
  }
  catch (double d) {
    cout << "Nivel 0. double : " << d << endl;
  }
}
```

În exemplul precedent, în cazul în care valoarea lui *n* este 3, se generează o excepție de tip `double`. Ea nu poate fi tratată complet la nivelul 1 al contextului, iar handlerul implicit o retransmite spre blocul imediat exterior. La acest nivel există un handler de tratare şi se afişează valoarea transmisă de excepție.

13.3 Excepții, constructori şi destructori

Principalul dezavantaj al utilizării funcțiilor satandard `setjmp` şi `longjmp` îl constituie faptul că acestea nu permit distrugerea obiectelor aflate pe stivă în momentul saltului spre funcția exterioară. După cum s-a precizat, mecanismul de tratare a excepțiilor limbajului C++ permite rezolvarea acestei probleme.

Toate obiectele complet create în blocul de unde s-a generat o excepție, sunt distruse în mod implicit de către compilator în cadrul handlerului de tratare a excepției respective (unde s-a recepționat excepția).

Exemplu. Programul următor utilizează clasa *Object* pentru a afişa momenele de creare şi distrugere ale obiectelor.

```
#include <iostream>
#include <string>
using namespace std;

class Object {
  string name;
public:
  Object(string n): name(n) {
    cout << "Constructor pentru obiectul " << n << endl;
  }
  Object(Object const &o): name(o.name + " (copy)")
  {
    cout << "Constructor copiere pentru obiectul "
         << name << endl;
  }
  ~Object() {
    cout << "Destructor pentru obiectul " << name <<
endl;
  }
  void f() {
    Object toThrow("'local obj'");
    cout << "Functia f din obiectul " << name << endl;
    throw toThrow;
  }
```

```
  void hello() {
     cout << "Hello de la obiectul " << name << endl;
  }
};

void main() {
  Object out("'main obj'");
  try {
    out.f();
  }
  catch (Object o) {
    cout << "Tratare exceptie\n";
    o.hello();
  }
  cout << “Dupa clauza catch\n”;
}
```

Ieşirea programului anterior este următoarea:

```
Constructor pentru obiectul 'main obj'
Constructor pentru obiectul 'local obj'
Functia f din obiectul 'main obj'
Constructor copiere pentru obiectul 'local obj' (copy)
Constructor copiere pentru obiectul 'local obj' (copy) (copy)
Destructor pentru obiectul 'local obj'
Tratare exceptie
Hello de la obiectul 'local obj' (copy) (copy)
Destructor pentru obiectul 'local obj' (copy) (copy)
Destructor pentru obiectul 'local obj' (copy)
Dupa clauza catch
Destructor pentru obiectul 'main obj'
```

Observații.

1. Funcția *f* din clasa *Object* generează o excepție, astfel încât după mesajul din linia a treia, se crează prin copiere un obiect asociat excepției în cadrul instrucțiunii `throw` (mesajul din linia a patra).
2. Deoarece clauza `catch` se comportă în mod asemănător unei funcții, se mai crează (prin copierea obiectului generat de excepție) un obiect din clasa *Object* pentru parametrul *o* (mesajul din linia a cincea). Obiectul local se distruge (linia a şasea) şi execuția funcției *f* se termină, execuția programului continuând în clauza `catch` din funcția `main`.
3. După afişarea mesajului şi apelul funcției *hello*, corpul clauzei `catch` se termină şi se apelează destructorul pentru obiectul *o*. Tot acum se distruge şi obiectul asociat excepției, generat de instrucțiunea `throw` (mesajele din liniile 9 şi 10).
4. După terminarea funcției mai, se distruge şi obiectul *‘main obj’*.

Dacă se doraşte ca obiectul generat de excepție să nu fie copiat încă o dată pentru clauza `catch` (transfer prin valoare), se poate specifica în antetul clauzei o ***referință la clasa asociată excepției*** în locul clasei respective.

D exemplu, dacă în programul anterior, se modifică antetul clauzei `catch` astfel:

```
catch (Object o) {
```

ieşirea programului este următoarea:

```
Constructor pentru obiectul 'main obj'
Constructor pentru obiectul 'local obj'
Functia f din obiectul 'main obj'
Constructor copiere pentru obiectul 'local obj' (copy)
Destructor pentru obiectul 'local obj'
Tratare exceptie
Hello de la obiectul 'local obj' (copy)
Destructor pentru obiectul 'local obj' (copy)
Dupa clauza catch
Destructor pentru obiectul 'main obj'
```

Observație. Deşi este permis, în general nu este indicat ca în antetul unei clauze `catch` să se specifice un ***pointer la clasa asociată excepției***. În acest caz, în cadrul instrucțiunii `throw` trebuie specificat operatorul `new` pentru crearea unui obiect. Obiectul respectiv nu mai este distrus automat de către compilator, distrugerea lui trebuind să fie realizată explicit de către programator.

13.4 Clase şi ierarhii de clase pentru tratarea excepțiilor

Pentru determinarea unei clauze `catch` care se va executa dintr-o secvență de mai multe clauze, compilatorul determină prima clauză din secvență la care tipul parametrului se potriveşte cu tipul obiectului generat de excepție. Această potrivire nu trebuie însă să fie perfectă, deoarece se utilizează principalele reguli ale derivării claselor.

Din acest motiv, pentru mai multe excepții înrudite, se pot forma ierarhii de clase pentru tratarea lor, iar clasele dintr-o asemenea ierarhie pot apare în antetul clauzelor `catch`.

Un obiect al unei clase derivate, sau o referință spre un obiect al unei clase derivate, poate selecta un handler de tratare care are ca parametru un obiect al clasei de bază, sau o referință spre acesta.

Din acest motiv, în cazul unei secvențe de handlere, specificarea acestora trebuie făcută de la clasa cea mai de jos a ierarhiei spre clasa de bază.

Exemplu. Următorul program utilizează o ierarhie de trei clase pentru tratarea excepțiilor.

```
#include <iostream>
using namespace std;

class Avertisment {};
class Eroare: public Avertisment {};
```

```
class EroareFatala: public Eroare {};

void f() {
  int n;
  cout << "Nivelul erorii: ";
  cin >> n;
  switch (n) {
    case 1: throw Avertisment();
    case 2: throw Eroare();
    case 3: throw EroareFatala();
  }
}

void main() {
  try {
    f();
  }
  catch(Avertisment) {
    cout << "Tratare exceptie Avertisment" << endl;
  // Urmatoarele handlere sunt acoperite de precedentul
  }
  catch(Eroare) {
    cout << "Tratare exceptie Eroare" << endl;
  }
  catch(EroareFatala) {
    cout << "Tratare exceptie EroareFatala" << endl;
  }
}
```

Se observă faptul că indiferent de tipul excepției generate, se selectează întotdeuna primul handler. Versiunea corectă a funcției `main` este următoarea:

```
void main() {
  try {
    f();
  }
  catch(EroareFatala) {
    cout << "Tratare exceptie EroareFatala" << endl;
  }
  catch(Eroare) {
    cout << "Tratare exceptie Eroare" << endl;
  }
  catch(Avertisment) {
    cout << "Tratare exceptie Avertisment" << endl;
  }
}
```

O variantă des utilizată în cazul ierarhiilor de clase pentru tratarea excepțiilor o constituie polimorfismul şi ***utilizarea pointerilor la clasa de bază***. În acest caz, clasa de bază a ierarhiei este indicat să fie o clasă abstractă.

Deoarece se utilizează pointeri pentru transmiterea informaţiilor asupra excepţiei curente, trebuie în plus ca în cadrul handlerelor de tratare, obiectele să fie distruse în mod explicit cu ajutorul operatorului `delete`.

Exemplu. Programul următor defineşte o ierarhie de clase cu o clasă de bază abstractă. Funţia virtuală pură *ProcesareExceptie* este suprascrisă în toate clasele derivate pentru procesarea fiecărui tip de excepţie.

```
#include <iostream>
using namespace std;

class Exceptie {
protected:
  char m[20];  // Mesajul exceptiei
public:
  virtual ~Exceptie() { }
  virtual void ProcesareExceptie() = 0;
friend ostream& operator<<(ostream& os, Exceptie& e)
  { return os << e.m; }
};

class Avertisment: public Exceptie {
public:
  Avertisment(char* s) {
    strcpy(m, s);
  }
  void ProcesareExceptie() {
    cout << m;
    exit(1);
  }
};

class Eroare: public Exceptie {
public:
  Eroare(char* s) {
    strcpy(m, s);
  }
  void ProcesareExceptie() {
    cout << m;
    exit(1);
  }
};

class EroareFatala: public Exceptie {
public:
  EroareFatala(char* s) {
    strcpy(m, s);
  }
  void ProcesareExceptie() {
    cout << m;
```

```
    exit(1);
  }
};

void f() {
  int n;
  cout << "Nivelul erorii: ";
  cin >> n;
  switch (n) {
    Exceptie* e;
    case 1:
      e = new Avertisment("Avertisment !");
      throw e;
    case 2:
      e = new Eroare("Eroare !!");
      throw e;
    case 3:
      e = new EroareFatala("Eroare fatala !!!");
      throw e;
  }
}

void main() {
  try {
    f();
  }
  catch(Exceptie* e) {
    e->ProcesareExceptie();
     delete e;
  }
}
```

În vadrul programelor se pot utiliza de asemenea ***clasele predefinite ale bibliotecii standard*** C++, referitoare la excepții. Această ierarhie are la bază clasa `exception`. Poate fi utilizată funcția publică `what()` a clasei exception, pentru afişarea unui text ce descrie eroarea.

Din clasa `exception` sunt derivate două clase, `logic_error`, pentru raportarea erorilor detectabile înainte de execuția programului şi `runtime_error`, pentru raportarea erorilor din timpul execuției programului.

Principalele clase derivate din `logic_error` sunt următoarele:

- `domain_error` – raportează violarea unei precondiții
- `invalid_argument` – argument invalid pentru o funcție
- `length_error` – un obiect cu o lungime mai mare decât NPOS (valoarea maximă reprezentabilă)
- `out_of_range` – în afara domeniului
- `bad_cast` – operație `dynamic_cast` eronată

Principalele clase derivate din `runtime_error` sunt următoarele:

- `range_error` – violarea unei postcondiții
- `overflow_error` – operație aritmetică eronată (depășire)
- `bad_alloc` – eroare de alocare

13.5 Supraâncărcarea operatorilor `new` şi `delete`

Supraâncărcarea acestor operatori este descrisă în acest capitol, deoarece ea presupune în general utilizarea excepțiilor. Versiunile inițiale ale operatorului `new` returnau pointerul NULL în cazul în care operația de alocare a memoriei a eşuat. Versiunile curente utilizează excepțiile pentru tratarea acestui caz, deşi majoritatea compilatoarele limbajului C++ suportă ambele stiluri.

Operatorii descrişi în capitolul 9 pot fi supraâncărcați pentru clasele definite de programatori. Operatorii `new` şi `delete` pot fi supraîncărcați atât pentru clase definite de utilizatori, cât şi la nivel global. În plus, se pot supraâncărca şi operatorii ce lucrează cu tablouri (`new[]` şi `delete[]`).

Supraâncarcarea operatorilor `new` şi `delete` trebuie realizată cu atenție, în general trebuind respectate semantica şi modul de operare al operatorilor predefiniți ai limbajului.

A. Tratarea excepțiilor în cazul în care alocarea memoriei eşuează

Principala sarcină a operatorului `new` este de a aloca o zonă de memorie şi de a returna adresa de început a acesteia. Rezultă că un asemenea operator trebuie să posede cel puțin un parametru de tipul `size_t` şi să returneze o valoare de tipul `void*`.

Versiunile moderne ale compilatoarelor C++ utilizează însă excepțiile în cazul în care nu se poate aloca zona de memorie cerută. În acest caz, se apelează o funcție specifică, numită ***handler de tratare a erorii de alocare***. Aceasta poate genera o excepție de tipul `bad_alloc`, sau generarea excepției se poate face chiar de către operator.

Deşi în realitate este mai complicată, principial structura unui operator `new` poate avea forma următoare:

```
void operator new(size_t size) {
  // se incearca alocarea a size octeti de memorie
  if ( /* alocarea s-a efectuat cu succes */ )
    return ( /* pointer la adresa de inceput a zonei */ );
  set_new_handler(handler);
  (*handler)();
  throw bad_alloc();
}
```

S-a notat cu `handler` un pointer la o funcție handler, forma `void()`. Acestea reprezintă handlerele de tratare a erorilor de alocare. Funcția predefinită `set_new_handler` permite setarea funcției ce urmează să fie handlerul de tratare al erorilor de alocare.

În principal, scopul handlerelor de alocare este cel de a colecta zonele de memorie care nu se mai utilizează în program (`garbage collection`), dar aceasta este o operație dificilă.

În mod uzual, handlerul de tratare a erorilor de alocare eliberează o anumită zonă de memorie şi afişează un mesaj de atenționare. Pentru aceasta, se utilizează o anumită zonă de memorie alocată dinamic la începutul programului, care este accesibilă din handler şi poate fi eliberată.

Exemplu. Programul următor utilizează funcția `set_new_handler`, pentru a seta un handler propriu de tratare a erorilor de alocare.

```
#include <cstdio>
#include <new>
using namespace std;

// Bloc de memorie utilizat de functia printf pentru
afisarea
// mesajului de eroare
char * mess = new char[128];

void my_new_handler() {
  // Eliberarea blocului failsafe
  delete failsafe;
  printf( "Alocare esuata\n" );
}

void main() {
  // Declararea noului handler
  set_new_handler(my_new_handler);
  // Pointer la un bloc de memorie
  int* buff;
  for (double k=0; ; k++)
    buff = new int[1024];
  // ...
}
```

B. Supraâncărcarea operatorilor globali `new` şi `delete`

Operatorii `new` şi `delete` pot fi supraâncărcați şi la nivel global, în afara oricăror clase. În acest caz însă ei acperă operatorii cu acelaşi nume din spațiul standard al numelor, astfel încât ei nu mai pot fi utilizați. De aceea, trebuie acordată o atenție sporită în acest caz.

În mod uzual acest mod de supraâncărcare este justificat atunci când metoda de alocare a memoriei utilizată de `new` şi `delete` nu este satisfăcătoare pentru anumite aplicații, în special cele de timp real, sau în cazul în care poate apare fenomenul de fragmentare al memoriei.

Pentru alocarea memoriei se pot utiliza funcțiile standard de alocare: `malloc`, `calloc`, `realloc`, iar pentru dealocare funcția `free`. Operatorul `delete` are un parametru de tipul `void*` şi eliberează memoria ocupată anterior de pointerul specificat ca parametru.

În cazul în care se utilizează pentru crearea / distrugerea unor obiecte, în plus aceşti operatori apelează şi constructori / destructori pentru clasele respective. Apelul acestora însă nu poate fi controlat prin program, el fiind realizat direct de către compilator.

Exemplu. Următorul proram utilizează supraâncărcarea operatorilor globali `new` şi `delete`.

```
#include <cstdio>
#include <cstdlib>
using namespace std;

void* operator new(size_t sz) {
  printf("Alocare %d octeti\n", sz);
  void* m = malloc(sz);
  if(!m) puts("Eroere de alocare\n");
  return m;
}

void operator delete(void* m) {
  puts("Dealocare");
  free(m);
}

class A {
  int i[100];
public:
  S() { puts("Constructor A"); }
  ~S() { puts("Destructor A"); }
};

void main() {
  puts("Alocare/dealocare int");
  int* p = new int(47);
  delete p;
  puts("Alocare/dealocare A");
  A* a = new A;
  delete s;
  puts("Alocare/dealocare A[5]");
  A* sa = new A[5];
  delete []sa;
}
```

S-au utilizat funcţiile `printf` şi `puts`, deoarece la crearea obiectelor `cin`, `cout` şi `cerr` compilatorul foloseşte operatorii `new` şi `delete`.

C. Supraâncărcarea operatorilor `new` şi `delete` pentru clase de obiecte

D. Supraâncărcarea operatorilor `new[]` şi `delete[]`