Excepții

Tratarea unei erori constă în detectarea erorii şi apelarea componentei software însărcinată cu rezolvarea acestei situații. Mecanismul excepțiilor înlătură necesitatea de a verifica de fiecare dată starea unui apel de funcție şi separă codul de tratare a erorilor de codul "normal"; indiferent de numărul de apeluri ale unei funcții, eventualele erori sunt tratate o singură dată, într-un singur loc.

I. Metode tradiționale de tratare a erorilor

În cele ce urmează nu discutăm despre situațiile în care, la detectarea unei erori, dispunând de toate informațiile necesare, putem trata eroarea imediat; ne vom referi la situațiile când nu există suficiente informații pentru a trata eroarea în contextul curent şi trebuie să plasăm informațiile disponibile despre eroare într-un context superior, în speranța că acolo există date suplimentare ce permit tratarea erorii. De exemplu, o funcție de alocare de memorie nu poate şti de ce nu mai există memorie disponibilă în sistem; cauza poate fi o eroare hardware, dar şi numărul foarte mare de aplicații deschise. Această funcție nu poate să facă altceva decât să anunțe producerea neplăcutului eveniment în speranța că cineva de la un nivel mai înalt poate "face rost" de mai multă memorie.

Există 4 politici general acceptate de tratare a erorilor în limbajul C. Prima dintre ele constă în faptul că funcțiile returnează informații despre eroare sau, dacă valoarea returnată nu poate fi utilizată în această manieră, setează un cod de eroare într-o variabilă globală. Standardul C, pune la dispoziție, prin header-ul <errno.h>, variabila globală de tip int errno şi funcția perror(). Aceasta însemnă că, după fiecare apel de funcție, valoarea returnată de aceasta şi/sau variabila errno trebuie verificate; după tratarea erorii, errno trebuie resetată explicit. Câți dintre noi verifică valoarea returnată de printf() (adică numărul de caractere efectiv afişate)? Cu alte cuvinte, această politică raportează eroarea dar nu garantează tratarea ei! Mai mult, într-un mediu cu mai multe fire de execuție valoarea lui errno poate fi modificată de un alt fir de execuție înainte de a fi consultată în firul curent.

A doua politică constă în utilizarea sistemului (puțin cunoscut) de tratare a semnalelor pe care standardul C îl oferă. Funcția raise() este utilizată pentru a genera evenimente (semnale); la producerea acestor evenimente reacționează nişte funcții numite handler-e, funcții setate cu signal(). Utilizatorul unei librării ce utilizează acest mecanism trebuie să înțeleagă şi să instaleze handler-ele corespunzătoare. Utilizarea mai multor astfel de librării poate genera conflicte între numerele semnalelor.

A treia politică constă în utilizarea funcțiilor de deplasare ne-locală pe care librăria C-standard le pune la dispoziție: setjmp() şi longjmp(). Cu setjmp() se salvează o stare "bună", stabilă a programului; dacă programul atinge o stare instabilă (se detectează o eroare) atunci se restaurează cu longjmp() acea stare "bună".

```
//exemplul 1
#include <cstdlib>
#include <iostream>
#include <csetjmp>
using namespace std;

class X{
public:
        X(){ cout << "ctor" << endl;}
        ~X(){ cout << "dtor" << endl;}
};

jmp_buf stare;              //variabilă în care salvăm starea programului

void f(){
        X o;
        static int contor = 0;
        ++contor;
        cout << "Apelul nr. " << contor << " al lui f()" << endl;
        if (contor <= 5)
                longjmp(stare, contor);      //salt ne-local; contor anunță de unde venim
}

int main(){
        int label;
        label = setjmp(stare);
        if (label = = 0){              //primul apel setjmp() salvează starea curentă si returnează 0
                cout << "am salvat starea!" << endl;
                f();
        }
        else{                          //următoarele apeluri setjmp() returnează al doilea argument al lui longjmp()
                cout << "am venit de la " << label << endl;
                f();
        }
        return 0;
}
```

Acest program afişează următoarele mesaje:

```
//output-ul programului de la exemplul 1
am salvat starea
ctor
Apelul nr. 1 al lui f()
am venit de la 1
ctor
Apelul nr. 2 al lui f()
am venit de la 2
ctor
Apelul nr. 3 al lui f()
am venit de la 3
ctor
Apelul nr. 4 al lui f()
am venit de la 4
ctor
Apelul nr. 5 al lui f()
am venit de la 5
ctor
Apelul nr. 6 al lui f()
dtor
Press any key to continue . . .
```

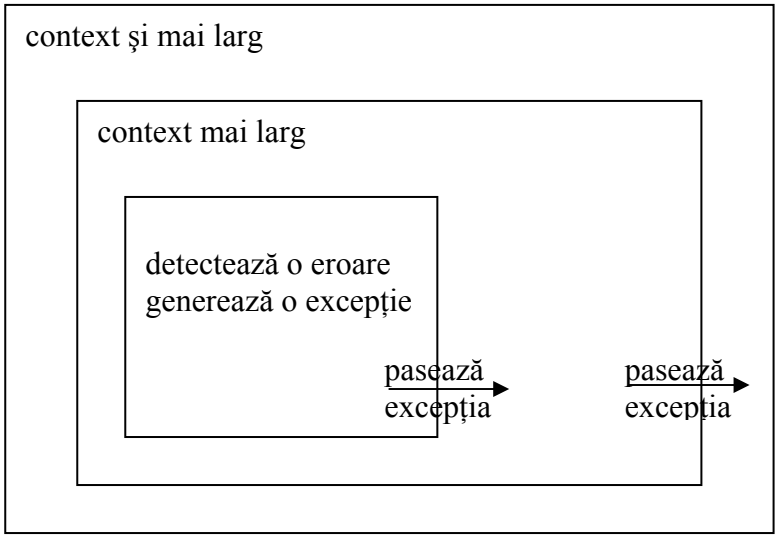
Apelul longjmp() aduce programul în starea imediat următoare apelului setjmp(); acesta se execută de mai multe ori si returnează 0 sau al doilea argument al lui longjmp() . Se observă imediat că, la fiecare apel al funcției f(), un obiect local X este creat (apelul constructorului) dar acesta nu este distrus (apelul destructorului lipseşte), ceea ce înseamnă că resursele achiziționate nu sunt eliberate! Acesta deoarece setjmp() şi longjmp() nu sunt "conştiente" de modelul obiect din C++[1].

A patra politică este cea mai simplă: terminarea imediată a programului în cazul detectării unei erori[2]. Librăria standard C pune la dispoziție la dispoziție funcțiile abort() (care termină **imediat** programul) şi exit() (care goleşte buffer-ele, închide fişierele deschise şi termină programul). Ele suferă însă de acceaşi problemă: nu distrug obiectele locale şi deci nu eliberează corect resursele, ceea ce poate produce erori ireparabile precum ştergerea unor fişiere pe hard-disk sau coruperea unei baze de date!

II. Tratarea erorilor prin mecanismul excepțiilor

Deoarece politicile tradiționale de tratare a erorilor din C nu sunt potrivite într-un mediu orientat-obiect, C++ a trebuit să introducă un mecanism specific de tratare a erorilor: excepțiile. Acestea au urmărit satisfacerea următoarele scopuri:

- la producerea unei erori, controlul execuției este transferat automat la punctul de tratare a erorii;
- dacă eroarea nu este tratată local (contextul curent), înainte de a pasa informațiile despre eroare către un nivel superioar (context mai larg) se distrug obiectele locale, eliberându-se resursele; dacă eroarea nu poate fi tratată nici în noul context, acest proces se reia şi informațiile despre eroare sunt transmise până când se ajunge într-un context care permite tratarea erorii;
- eliberarea programatorului de necesitatea de verifica, de fiecare dată, valoarea returnată de o funcție şi/sau variabila errno; ca o consecință imediată, separarea codului de tratare a erorilor de codul "normal".



O *excepție* este o instanță fie a unui tip predefinit, fie a unui tip definit de utilizator; în general, pentru a transmite unui context superior cât mai multe informații despre eroarea produsă se utilizează clase, astfel că în cele ce urmează vom vorbi despre obiecte excepții. Deoarece o excepție trebuie transmisă între contexte diferite de execuție, ea va fi un **obiect independent de cursul normal de execuție** al unui program. La generarea unei excepții, o funcție **returnează efectiv un obiect excepție**, chiar dacă acesta nu este de tipul pe care funcția trebuia să-l returneze; diferă însă, față de funcțiile obişnuite, unde este returnat obiectul excepție!

III. Componentele mecanismului de tratare a excepţiilor

Mecanismul de tratare a excepţiilor este acea componentă software însărcinată cu **transferul controlului execuţiei programului** de la punctul în care este generată o excepţie la punctul în care excepţia este gestionată. El este alcătuit din:

a) Blocul try

Un bloc try este un bloc scop obişnuit, precedat de cuvântul cheie try; el conţine o secvenţă de cod care *poate* genera excepţii. Astfel, dacă în C fiecare apel de funcţie este înconjurat de cod care să verifice valoarea returnată de acea funcţie, în C++ apelurile de funcţii sunt împachetate în blocuri try, nemaifiind necesară verificarea erorilor. În consecinţă, codul "normal" este mai uşor de citit şi de înţeles deoarece el este separat fizic de codul de tratare a erorilor.

```
try{
        //cod care poate genera excepţii (cod "normal")
}
```

b) Secvenţa de handler-e asociate blocului try

Bineînţeles, dacă în blocul try este generată o excepţie, aceasta trebuie să ajungă undeva; de aceea, blocul try este urmat de unul sau mai multe handler-e, marcate prin instrucţiuni catch. Prezenţa mai multor handler-e permite tratarea în mod diferit a diverselor tipuri de excepţii:

```
//exemplul 2
try{
        //cod care poate genera excepţii (cod "normal")
}
catch (type1& id){
        //tratarea unei excepţii de tipul type1
}
catch(type id){
        // tratarea unei excepţii de tipul type2
}
```

Fiecare handler se comportă ca o funcţie care recepţionează un singur argument: obiectul excepţie. Se observă că obiectul excepţie poate fi recepţionat de handler atât prin valoare, cât şi prin referinţă.

Dacă în blocul try este generată o excepţie, mecanismul de tratare a excepţiei "pleacă în căutarea" primului handler care are argumentul de un tip ce se potriveşte

cu tipul excepției. Dacă un astfel de handler este găsit, clauza catch care se potriveşte (şi numai ea – nu faceți confuzie cu switch!) este executată şi excepția se consideră a fi tratată (prinsă). Clauzele catch sunt verificate în ordinea apariției lor în codul sursă.

Tipul excepției determină handler-ul care o va prinde. Regulile de potrivire a tipului pentru obiectele excepție şi handler-e sunt mai restrictive decât în cazul supraîncărcării funcțiilor, permitând doar un set redus de conversii. Astfel, o excepție de tipul E se potriveşte la un handler care prinde T sau T& dacă:

- T şi E sunt de acelaşi tip (se ignoră const şi volatile), sau
- T este o bază publică neambiguă a lui E.

Dacă E şi T sunt pointeri, atunci excepția se potriveşte la handler dacă:

- T şi E sunt de acelaşi tip, sau
- E referă un obiect care este derivat public şi neambiguu din clasa obiectului spre care referă T.

Un handler tablou-de-T sau funcție-ce-returnează-T este transformat, respectiv, în pointer-la-T sau pointer-la-funcție-ce-returnează-T.

Deoarece handler-ele pentru clasele excepție derivate pot prinde obiectele excepție din clasele de bază, rezultă necesitatea de a verifica la execuție tipul unui obiect excepție; de aici, o strânsă legătură în excepții şi RTTI.

c) Desfăşurarea stivei (stack unwinding)

La generarea unei excepții, mecanismul de tratare a excepției caută în scopul curent un handler corespunzător. Dacă un astfel de handler nu există, se iese din scopul curent şi se intră în scopul blocului următor în lanțul de apeluri (adică în funcția sau blocul try care împachetează perechea try{}catch care a eşuat în prinderea excepției). Acest proces continuă până când, la un anumit nivel în lanțul de apeluri, un handler va prinde excepția. În acest punct excepția se consideră a fi tratată, căutarea se opreşte iar stiva a fost desfăşurată: obiectele locale construite pe calea de la punctul în care excepția a fost generată până la handler-ul care a prins-o au fost distruse (destructorii acestor obiecte au fost apelați)!

Dacă totuşi nu există nici un handler care să prindă excepția, aceasta se consideră a fi netratată. O excepție netratată apare şi dacă o nouă excepție este generată **înainte** ca excepția curentă să fie prinsă de un handler. Dacă există o excepție netratată, se apelează automat funcția standard terminate(), declarată în header-ul <exception>. Comportamentul implicit al acestei funcții este acela de a termina programul.

Standardul C++ garantează **desfăşurarea stivei doar dacă excepția a fost tratată**; depinde de implementare dacă o excepție netratată provoacă sau nu desfăşurarea stivei. Pentru a ne asigura că stiva este întotdeauna desfăşurată, putem include în main() o expresie catch-all:

```
//exemplul 3
int main(){
        try{
                //implementarea funcției main()
        }
        catch (std::exception& e){
                //tratează excepțiile planificate
        }
        catch (...){                    //catch-all
                //asigură desfăşurarea stivei în cazul producerii unei excepții neplanificate
        }
        return 0;
}
```

d) Expresiile throw/rethrow

O excepție poate fi generată doar de către o expresie throw; prin urmare, blocul try (sau o funcție apelată din blocul try) trebuie să conțină o astfel de expresie. O expresie throw este similară unei instrucțiuni return şi constă din instrucțiunea throw şi dintr-un operand:

```
try{
        throw 5;            //5 este atribuit lui n în handler
}
catch (int n){}
```

O expresie rethrow indică re-generarea excepției tratate şi este formată dintr-o instrucțiune throw fără operand. Dacă nu există o excepție tratată, expresia rethrow apelează terminate()).

e) Obiectul excepție

O excepție poate fi de un tip predefinit, precum int sau char*, dar poate fi şi o instanță a unei clase, cu date şi funcții membru, putând astfel oferi handler-ul care prinde excepția mai multe informații şi mai multe opțiuni de tratare a erorii produse.

O excepție poate fi pasată handler-ului corespunzător prin valoare sau prin referință. Standardul C++ nu specifică **cum** anume şi **de unde** se alocă memorie obiectelor excepție; se specifică însă că **nu** li se alocă memorie din free store. De aceea, o tehnică uzuală de implementare constă în gestionarea unei stive dedicate. Dacă o excepție este transmisă prin valoare, în contextul handler-ului care prinde excepția este construită o **copie** a obiectului excepție. Daca o excepție este transmisă prin referință, handler-ul care a prins excepția recepționează o referință la obiectul excepție, ceea ce asigură şi comportamentul polimorfic.

Pasarea excepțiilor prin valoare este costisitoare deoarece un obiect excepție poate fi construit şi distrus repetat de câteva ori până se ajunge la handler-ul de tratare. Totuşi, generarea unei excepții presupune că programul se află într-o stare anormală; într-o astfel de situație, considerațiile legate de performanțe devin secundare, importantă fiind menținerea integrității aplicației.

f) Specificația de excepție

Nu sunteți obligați să vă informați clienții ce tipuri de excepții poate genera o funcție a voastră; ar fi însă necivilizat să nu o faceți! Cum ei vor dispune doar de librăriile compilate şi de fişierele header, C++ vă permite să specificați în prototipul funcțiilor ce tipuri de excepții poate genera o funcție; aceasta este specificația de excepție.

O specificație de excepție face parte din declarația unei funcții, dar nu şi din tipul ei (adică nu puteți supraîncărca funcții doar pe baza specificației de excepție)! Mai mult, declararea unui typedef care conține o specificație de excepție provoacă o eroare la compilare. Specificația de excepție apare imediat după lista argumentelor şi reutilizează cuvântul cheie throw, care este urmat între paranteze de o listă a tipurilor posibilelor excepții:

```
//exemplul 4
class tooSmall{};
class tooBig{};
class unUsual{};
void function1() throw(tooSmall, tooBig, unUsual);   //poate genera doar excepții de tipurile tooSmall, tooBig, unUsual
void function2() throw();                            //nu generează excepții
void function3();                                    //poate genera orice tip de excepție!
typedef void (*PF)() throw(tooSmall);                //eroare la compilare
```

Ce se întâmplă dacă o funcție generează o excepție de un tip nespecificat în specificația sa de excepție? Mecanismul de tratare a excepțiilor detectează această situație şi apelează funcția standard unexpected() (declarată în <exception>). Cum o violare de specificație de excepție este probabil un bug, comportamentul implicit al funcției unexpected() este de a apela terminate().

Specificațiile de excepție se verifică doar la execuție. Din acest motiv compilatorul ignoră deliberat cod care pare, în mod clar, a viola specificațiile de excepție:

```
//exemplul 5
void function2() throw(){
        function3();                                 //poate genera orice tip de excepție
}
```

Deşi function2() promite să nu genereze excepții, dacă function3() ar genera o excepție, aceasta s-ar propaga prin function2(), evident încălcând astfel garanția dată în specificația de excepție. Totuşi, acest cod este legal pentru compilator deoarece specificațiile de excepție se verifică şi se impun la execuție de către mecanismul de tratare a excepțiilor. Dacă nu ar fi aşa, programatorul ar fi forțat să împacheteze apelul lui function3() într-un bloc try, deşi el poate că ştie că function3() nu va genera niciodată vreo excepție (este cazul evident al funcțiilor din librăria standard C). Cu alte cuvinte, C++ are încredere în programator!

C++ impune concordanța specificației de excepție pentru funcțiile membru din clasele derivate: funcțiile virtuale supraîncărcate într-o clasă derivată trebuie să posede o specificație de excepție identică sau mai restrictivă decât specificația de excepție a funcției virtuale din clasa de bază.

```
//exemplul 6
class baseException{};
class derivedException: public baseException{};
class otherException{};
class A{
        public:
        virtual void f() throw (baseException);
```

```
        virtual void g() throw (baseException);
        virtual void h() throw (derivedException);
        virtual void i() throw (derivedException);
        virtual void j() throw (baseException);
};
class B: public A{
        public:
        virtual void f() throw (derivedException);              //ok
        virtual void f() throw (otherException);                //eroare de compilare
        virtual void g() throw (derivedException);              //ok
        virtual void i() throw (baseException);                 // eroare de compilare
        virtual void j() throw (baseException, otherException); // eroare de compilare
};
```

IV. Blocuri try la nivel de funcție

Un bloc try la nivel de funcție este o funcție al cărei corp constă dintr-un bloc try şi handler-ele asociate. De exemplu, pentru a ne asigura că stiva este întotdeauna desfăşurată, putem scrie main() astfel:

```
//exemplul 7
int main() try{
        //cod main()
}
catch(...){}        // orice excepție este prinsă!
```

Blocurile try la nivel de funcție sunt utile mai ales pentru constructori, deoarece permit prinderea excepțiilor generate în lista de inițializare. Dacă lista de inițializare a unui constructor generează o excepție, construcția obiectului se opreşte. Handler-ul unui bloc try la nivel de funcție nu poate executa return; eventual, se poate ieşi cu throw, permițând generarea unei excepții conforme cu specificația de excepție.

```
//exemplul 8
class E{};
class C{
        std::string str;
        public:
                C::C(const std::string& s) throw (E) try : str(s){
                //corpul constructorului
                }
        catch(...){
                //tratează excepțiile generate în lista de inițializare sau în corpul constructorului
                //inițializarea lui str ar putea genera std::bad_alloc, în contradicție cu specificația de excepție
                throw E();          //pentru a genera o excepție conformă cu specificația dată
                }
};
```

V. Excepții în constructori

C++ nu apelează destructorul unui obiect care nu au fost construit complet (adică pentru care constructorul nu şi-a terminat execuția). Este firesc să fie aşa: dacă obiectul nu este complet construit, de unde să ştie destructorul în ce stare se află

obiectul şi ce trebuie făcut? Acesta conduce însă la situații precum cea din exemplul următor:

```
//exemplul 9
class Fotografie{
        public :
        Fotografie(const string& nume_fisier);
        ...
};

class Plan{
        public:
        Plan(const string& nume_fisier);
        ...
};

class Anunt {
        public:
        Anunt (const string& n, const string& fisier_foto="", const string fisier_plan=""): nume(n), f(0), p(0){
                if (fisier_foto != "") f = new Fotografie(fisier_foto);
                if (fisier_plan != "") p = new Plan(fisier_plan);          //(*)
        }
        ~Anunt (){
                delete f;
                delete p;
        }
        private:
        string nume;
        Fotografie* f;
        Plan* p;

};
```

În exemplul anterior, ultimele două argumente ale constructorului clasei Anunt sunt opționale, așa că f şi p sunt inițializați cu 0. Dacă argumentele sunt furnizate, lui f şi lui p li se atribuie valori corespunzătoare. Ce se întâmplă însă dacă pe linia marcată (*) operatorul new sau Plan() generează o excepție? Construcția obiectului Anunt încetează; obiectul nu este construit complet, aşa că destructorul său nu se apelează. Cine distruge atunci obiectul Fotografie referit de f şi evită pierderea de memorie (memory leak)? Singurul care o poate face este însuşi constructorul!

Constructorii trebuie proiectați astfel încât să fie capabili să "facă curățenie" după ei dacă execuția lor este întreruptă de apariția unei excepții. De obicei, acesta însemnă prinderea tuturor excepțiilor posibile, execuția unei secvențe de cod pentru eliberarea resurselor şi repropagarea excepției.

```
//exemplul 10
        Anunt::Anunt (const string& n, const string& fisier_foto="", const string fisier_plan=""): nume(n), f(0), p(0){
                try{
                        if (fisier_foto != "") f = new Fotografie(fisier_foto);
                        if (fisier_plan != "") p = new Plan(fisier_plan);
                }
                catch(...){          //catch-all
```

```
                delete f;
                delete p;
                throw;          //rethrow
        }
}
```

O altă idee este să transformăm pointerii f şi p în obiecte pointer, utilizând clasa şablon auto_ptr<> din librăria standard.

```
//exemplul 11
class Anunt {
        public:
        Anunt (const string& n, const string& fisier_foto="", const string fisier_plan=""): nume(n), f(0), p(0){
                if (fisier_foto != "") f = new Fotografie(fisier_foto);
                if (fisier_plan != "") p = new Plan(fisier_plan);          //(*)
        }
        ~Anunt (){}
        private:
        string nume;
        auto_ptr<Fotografie> f;
        autoptr<Plan> p;

};
```

Deoarece f este obiect (complet construit), el este distrus automat dacă apare o excepție in linia (*). Mai mult, f şi p sunt distruşi automat odată cu obiectul Anunt, iar destructorii lor eliberează automat memoria alocată!

VI. Excepții în destructori

Propagarea unei excepții în afara destructorului este periculoasă şi nerecomandată. De ce? Destructorul poate fi invocat de mecanismul de desfăşurare a stivei, în urma producerii unei excepții; dacă destructorul generează o nouă excepție, acesta va fi netratată şi se va apela terminate(). Un alt motiv ar fi că, dacă excepția nu este prinsă, destructorul îşi opreşte execuția în punctul în care excepția a fost generată şi, foarte probabil, nu eliberează resursele. Din aceste motive, un destructor care permite propagarea unei excepții este o greşeală de design (de proiectare a clasei).

Dacă totuşi chiar trebuie să generați o excepție din destructor, trebuie testat dacă nu cumva o altă excepție este procesată. O excepție este prinsă dacă s-a intrat într-un handler sau dacă a fost apelată unexpected(). Pentru a verifica dacă excepție este procesată, se utilizează funcția uncaught_exception() din <stdexcept>:

```
//exemplul 12
class FileException{};
File::~File() throw (FileException){
        if (close(handler) != succes)          //nu am reuşit să închidem fişierul
```

```
        {
                if (uncaught_exception() == true)
                        return;                 //nu putem genera excepție
                else throw FileException();
        }
        return;   //succes
}
```

VII. Excepții standard

Headerul <stdexcept> conține definițiile claselor excepție standard. Clasa de bază pentru toate excepțiile generate de librăria standard este exception; aceasta conține funcția virtuală what(), ce returnează o descriere verbală a excepției sub forma unui şir de caractere const char*. Din exception sunt derivate logic_error, care raportează erori logice (adică erori care ar fi trebuit detectate înainte de compilare), runtime-error, care raportează erori de execuție (adică erori detectabile doar la execuție) şi ios::failure, care raportează erori pentru I/O stream-uri.

Excepții derivate din logic_error	
domain_error	violarea unei precondiții
invalid_argument	argument invalid pentru o funcție
length_error	încercarea de a crea un obiect a cărui dimensiune este mai mare decât NPOS (cea mai mare valoare reprezentabilă de tipul size_t)
out_of_range	depăşire index
bad_cast	expresie dynamic_cast<> invalidă
bad_typeid	raportarea unui pointer p NULL într-o expresie typeid(*p)

Excepții derivate din runtime_error	
range_error	violarea unei postcondiții
overflow_error	overflow aritmetic
bad_alloc	eşec de alocare de memorie
bad_exception	violarea specificației de excepție a unei funcții

VIII. Modificarea comportamentului lui unexpected() şi terminate()

Atunci când mecanismul de tratare a excepţiilor detectează o violare a unei specificaţii de excepţie, se apelează funcţia standard unexpected(). Aceasta este implementată prin intermediul unui pointer de tipul void (*pf)() (funcţie fără argumente şi care nu returnează); implicit, acest pointer referă terminate(). Comportamentul lui unexpected() poate fi modificat cu set_unexpected(), care cere drept argument adresa unei funcţii fară parametri şi care nu returnează; set_unexpected() returnează valoarea anterioară a pointerului lui unexpected(), astfel încât acesta să poată fi salvat şi, eventual, restaurat.

```
//exemplul 13
#include <cstdlib>
#include <iostream>
#include <exception>
using namespace std;

class E{};

void my_unexpected(){
        cout << "a fost apelata unexpected()!" << endl;
        getchar();
        throw E();          //tip permis in specificarea de excepţie a lui function()
}

void function() throw(E){
        cout << "violare a specificatiei de exceptie pentru function()" << endl;
        getchar();
        throw "exceptie de tip const char*";
}

int main(){
        set_unexpected(my_unexpected);
        try{
                function();
        }
        catch(E& e){
                cout << "exceptia a fost prinsa!" << endl;
        getchar();
        }
}
```

În exemplul anterior, apelarea lui function() în main() generează o excepţie de tipul const char*; violându-se specificaţia de excepţie a lui function(), este apelată unexpected(), care, la rândul ei, apelează my_unexpected().

Modificarea comportamentului lui unexpected() este utilă pentru a putea genera un nou tip de excepţie, care să o substituie pe cea care a provocat violarea specificaţiei de excepţie. Dacă funcţia unexpected() generează o excepţie, căutarea handler-ului corespunzător reîncepe de la funcţia care a generat excepţia iniţială. În

cazul nostru, my_unexpected() generează o excepție de tip E, permisă în specificarea de excepție a lui function(); noua excepție se propagă prin unexpected() şi este apoi prinsă de handler.

Dacă my_unexpected() generează o excepție nepermisă în specificația de excepție a lui function(), dar specificația de excepție a lui function() admite tipul standard std::bad_exception, atunci unexpected() generează o excepție de acest tip.

```
//exemplul 14
#include <cstdlib>
#include <iostream>
#include <exception>
using namespace std;

class E{};
class EE{};

void my_unexpected(){
	cout << "a fost apelata unexpected()!" << endl;
	getchar();
	throw EE();			//tip nepermis de specificarea de exceptie a lui function()
}
void function() throw(E, std::bad_exception){
	cout << "violare a specificatiei de exceptie pentru function()" << endl;
	getchar();
	throw "exceptie de tip const char*";
}
int main(){
	set_unexpected(my_unexpected);
	try{
		function();
	}
	catch(E& e){
		cout << "exceptia a fost prinsa!" << endl;
		getchar();
	}
	catch(std::bad_exception& e){
		cout << "std::bad_exception a fost prinsa!" << endl;
		getchar();
}
}
```

Dacă my_unexpected() generează o excepție nepermisă în specificația de excepție a lui function(), iar specificația de excepție a lui function() nu admite tipul std::bad_exception, atunci unexpected() apelează direct terminate(), aşa cum se vede din exemplul următor:

```
//exemplul 15
#include <cstdlib>
#include <iostream>
#include <exception>
using namespace std;

class E{};
class EE{};
```

```
void my_terminate(){
	cout << "a fost apelata terminate()!" << endl;
	getchar();
}

void my_unexpected(){
	cout << "a fost apelata unexpected()!" << endl;
	getchar();
	throw EE();			//tip nepermis de specificarea de exceptie a lui function()
}

void function() throw(E){
	cout << "violare a specificatiei de exceptie pentru function()" << endl;
	getchar();
	throw "exceptie de tip const char*";
}

int main(){
	set_terminate(my_terminate);
	set_unexpected(my_unexpected);
	try{
		function();
	}
	catch(E& e){
		cout << "exceptia a fost prinsa!" << endl;
		getchar();
	}
}
```

Pentru o excepție netratată se apelează automat funcția standard terminate(). Şi această funcție este implementată prin intermediul unui pointer de tipul void (*pf)(); implicit, acest pointer referă abort(). Comportamentul lui terminate() poate fi modificat cu set_terminate(), aşa cum s-a văzut în exemplul anterior. Cum my_terminate() nu a terminat programul, terminate() a apelat imediat abort().

Deoarece terminate() apelează abort(), destructorii obiectelor alocate static nu sunt apelați! Utilitatea lui my_terminate() rezidă în posibilitatea de a apela explicit aceşti destructori înainte de a apela abort():

```
//exemplul 16
#include <cstdlib>
#include <iostream>
#include <exception>
using namespace std;

class E{};

class C{
	public:
	~C(){
		cout << "~C()" << endl;
	}
};

C o;				//obiect global
```

```
void my_terminate(){
        cout << "a fost apelata terminate()!" << endl;
        o.~C();             //apel explicit de destructor pentru un obiect global
        getchar();
        //abort();          //apel explicit abort()
}

void function() throw(E){
        cout << "violare a specificatiei de exceptie pentru function()" << endl;
        getchar();
        throw "exceptie de tip const char*";
}

int main(){
        set_terminate(my_terminate);
        try{
        function();
        }
        catch(E& e){
                cout << "exceptia a fost prinsa!" << endl;
                getchar();
        }
}
```

IX. Sfaturi

- Nu generați excepții dacă posedați suficiente informații pentru a trata imediat eroarea apărută!
- Dotați-vă întotdeauna funcțiile cu specificații de excepții!
- Nu utilizați excepțiile pentru a controla execuția programului; este ineficient (suprasarcinile induse de mecanismul de tratare a excepțiilor pot conduce la serioase penalități de execuție în programele mari) şi confuz pentru utilizator!

```
//exemplul 17
class Exit{};
int main(){
        int numar;
        cout << “introduceti un numar; cu 66 se termina: << endl;
        try{
                while(true){
                                cin >> numar;
                                if (numar == 66) throw Exit();
                                cout << “ati introdus: “ << numar << endl;
                }
        }
        catch (Exit& ){
                cout << “la revedere!” << endl;
        }
        return 0;
}
```

- Utilizați clasele excepție standardizate înainte de a vă crea propriile clase excepție!

- Dacă vă creați propriile clase excepție, derivați-le din std::exception şi supraîncărcați what()!
- Dacă creați o clasă excepție pentru a fi folosită doar de o anumită clasă, împachetați clasa excepție în definiția clasei care o utilizează!
- Utilizați ierarhiile de excepții pentru a clasifica diferitele tipuri de erori!
- Prindeți excepțiile prin referință, pentru a garanta polimorfismul! Nu uitați că prinderea excepțiilor prin valoare înseamnă obținerea unei copii a obiectului excepție. Dacă obiectul excepție este prins prin valoare de un handler al unei clase de bază, copia este automat convertită la tipul de bază, pierzându-se astfel eventualele informații suplimentare despre eroare!
- Feriți-vă de pierderile de memorie în constructori!!
- Nu propagați excepțiile în afara destructorilor!!!

Norocel PETRACHE

Bibliografie

- ISO/IEC 14882 – Programming Languages – C++
- Bjarne Stroustrup - The C++ Programming Language, ed. III
- Scott Meyers – More Effective C++
- Herb Sutter – Exceptional C++
- Danny Kalev - ANSI/ISO C++ Professional Programmer's Handbook

[1] Unele compilatoare C++ au extins longjmp() a.î. să curețe stiva şi să apeleze destructorii. Acest comportament este ne-standard, deci ne-portabil.

[2] Terminarea unui program poate fi acceptabilă în anumite condiții extreme sau în fazele depanare. Evident, anumite aplicații nu pot fi pur şi simplu oprite la apariția unei erori; ce s-ar întâmpla dacă programul computerului unui avion s-ar opri pentru că, din cauza condițiilor atmosferice, s-a pierdut temporar legătura cu turnul de control?