Obsah

Ošetrovanie výnimiek ... 2
Staré spôsoby ošetrovania výnimiek ... 2
OOP prístup k ošetrovaniu výnimiek ... 5
Vysielanie výnimky ... 5
Vysielanie inicializovaných objektov ... 6
Zachytávanie výnimky ... 7
Try blok ... 7
Rutina ošetrenie výnimky ... 7
Ukončenie a obnovenie ... 8
Vyhľadanie správnej obsluhy výnimky ... 9
Rozbalenie zásobníka ... 9
Porovnávanie výnimiek ... 9
Zachytávanie akejkoľvek výnimky ... 11
Znovuvyslanie výnimky ... 11
Nezachytené výnimky ... 11
Funkcia terminate ... 11
Funkcia set_terminate ... 12
Upratovanie ... 13
Správa prostriedkov ... 14
Všetko ako objekt ... 15
Try bloky na úrovni funkcie ... 17
Štandardné výnimky ... 19
Špecifikácia výnimky ... 20
Funkcia unexpected ... 20
Funkcia set_unexpected ... 21
Kedy *nepoužívať* špecifikáciu výnimiek ... 23
Programovanie s výnimkami ... 24
Typické použitie výnimiek ... 25
Kedy používať špecifikácie výnimiek ... 25
Začnime so štandardnými výnimkami ... 25
Vnárajme svoje vlastné výnimky ... 25
Používajme hierachie výnimiek ... 25
Zachytávajme odkazom, nie hodnotou ... 26
Vysielajme výnimky v konštruktoroch ... 26
Negenerujme výnimky v deštruktoroch ... 26
Réžia použitia výnimiek ... 26
Zhrnutie ... 27

Ošetrovanie výnimiek

V živote nejde vždy všetko tak ako by sme chceli. V programovaní sa neočakávaný alebo nezvyčajný stav nazýva ***výnimka***. Zdokonalenie zotavenia kódu po chybe je jedným z najmocnejších prostriedkov, ktorým zvýšime odolnosť kódu.

Neočakávané stavy sprevádzajú programovanie od samého počiatku, avšak pojem výnimka a jej ošetrovanie v tej dobe nebol známy. Programátori mali dosť čo robiť s udržiavaním systému v chode a nie sa ešte zaoberať nejakou teóriou zotavenia po chybe a ošetrovaním výnimiek. Programy riešili neočakávané udalosti mnohými spôsobmi. Počítačové programovanie sa však vyvíjalo veľmi rýchlo a s objavením sa kompilátorov vznikali rozsiahlejšie programy, čo viedlo k pokusom systematicky sa zaoberať chybami v štruktúrovaných systémoch.

Zaoberať sa výnimkami systematickým spôsobom bola výzva z mnohých dôvodov. Avšak skôr ako sa na tieto dôvody pozrieme, je dobré pozrieť sa na terminológiu a povedať si čo to vlastne výnimka je.

Keď program zistí výnimočný stav, informuje zvyšok systému tzv. ***generovaním*** alebo ***vyslaním*** výnimky. Niekde v programe by sa mal nachádzať kód, ktorý výnimku zachytí a spracuje. Pre výnimky platí:

1. Výnimka normálne reprezentuje nejakú ***zlú*** udalosť. Napríklad ak má nejaká nízko-úrovňová funkcia zapisovať dáta na disk, výnimku môže reprezentovať zlyhanie hardvéru – neúspech zápisu dát.

2. Výnimka predstavuje nezvyčajný a neočakávaný stav. Napríklad alokačná rutina pamäti považuje zlyhanie alokácie za výnimku, pretože jej volanie z programu očakáva, že požiadavka na pamäť bude rešpektovaná. Hoci by sme výnimku mohli indikovať i v opačnom prípade, t.j. pri úspechu získania požadovanej pamäti, takáto výnimka by bola zavadzajúca a koncepčne nesprávna.

3. Výnimky môžu byť spôsobené externými faktormi a faktormi prostredia. Napríklad pamäť je niečo, čo operačný systém poskytuje programom. Ak program pamäť vyčerpá, môže to byť spôsobené tým, že operačný systém nedokáže alokovať dostatok pamäti alebo pamäti je nedostatok, alebo oboje. Pretože výnimky môžu byť spôsobené i externými faktormi, program nedokáže vždy problém odstrániť a obnoviť pôvodný stav.

Program sa môže dostať do mnohých stavov, ktoré možno alebo nemožno považovať za výnimky. Zriedkavé udalosti možno považovať za výnimky, ale čo ak sú tieto udalosti očakávané? Napríklad čo objekty, ktoré sa uchovávajú v kontajneroch? Má sa generovať výnimka, ak sa ten istý objekt nájde v kontajneri viackrát? Závisí to od toho, či je dobré alebo zlé (čítaj správne alebo nesprávne), že sa objekt nachádza v kontajneri viackrát. Čo program v jednom prípade považuje za prípustné môže byť v inom prípade nežiadúce.

Staré spôsoby ošetrovania výnimiek

V jazyku C problém prístupu ku ošetrovaniu chýb by sme si mohli predstaviť ako problém väzby - užívateľ funkcie musí spájať kód na ošetrenie chyby tak tesne s funkciou, že táto sa stáva príliš nemotornou a ťažko sa používa.

Ošetrenie chyby je možné v situáciách, v ktorých presne poznáme čo máme robiť, pretože máme k dispozícii všetky potrebné informácie súvisiace s príčinou chyby. Malo by byť samozrejmé, že chybu ošetríme práve v tomto bode.

Problém nastane, keď nemáme dostatok informácií o celkovej situácií a chybovú informáciu potrebujeme presunúť do širšieho kontextu, kde sa takáto informácia nachádza.

Pred vyvinutím formalizovaného ošetrenia výnimiek, C++ programátori (ako i ostatní programátori) hľadali rôzne východiská ošetrovania chýb, neočakávaných stavov a porúch. Nasledovný zoznam vymenúva najbežnejšie spôsoby ošetrenia chýb a výnimiek.

1. **Celý program sa ukončí.**

   Toto je metóda, ktorú používa napríklad makro **assert** jazyka C. Ukončenie programu by malo byť považované za poslednú možnosť ošetrenia chýb, keď obnova (zotavenie) nie je možné, ťažké, riskantné alebo nespoľahlivé. V mnohých prípadoch môže byť ukončenie programu pri výskyte chyby nebezpečnejšie, než pokračovanie v jeho vykonávaní. Zoberme napríklad nukleárny reaktor, riadený programom, kde napríklad vznikne chyba delenia nulou. Poslednou vecou, ktorú by sme chceli urobiť, je program ukončiť. Výnimku treba ošetriť a program by mal byť schopný nejakým spôsobom pokračovať vo vykonávaní (najlepšie po varovaní operátora...).

2. **Funkcie, detekujúce výnimku, vrátia stavovú hodnotu, indikujúcu že nastalo niečo nesprávne.**

   Funkcia môže vracať 1 ak je úspešná, 0 ak je neúspešná. Na uchovanie chyby medzi volaniami funkcie by sme mohli použiť globálnu premennú. Avšak po sebe idúce volania nízko-úrovňových funkcií by mohli spôsobiť ich zlyhanie, ak globálna chybová premenná indikuje predchádzajúci problém. Tento spôsob používa štandardná vstupno/výstupná knižnice jazyka C, kde chybový kód uchováva globálna premenná **errno**.

   Problém tohto spôsobu spočíva v tom, že hoci robí dobrú prácu pri detekcii chyby, nerobí nič pre ich ošetrenie alebo obnovenie pôvodného stavu. Volajúce funkcie sa spoliehajú na kontrolu návratových kódov funkcií a chybové premenné. Tento spôsob ošetruje výnimky ako dvojzložkový proces: nízko-úrovňový kód detekuje chyby a vysoko-úrovňové funkcie chyby ošetrujú.

   Dvojzložkový proces je bránou k problémom. Aj keď nízko-úrovňový kód urobí svoju prácu a správne detekuje všetky chyby, neexistuje záruka, že systém, používajúci tento kód ošetrí chyby správne (alebo či ich vôbec ošetrí). Naviac ak každé volanie nízko-úrovňovej funkcie treba kontrolovať na chyby, programy by mohli byť extrémne rozsiahle, pričom väčšia časť kódu by bola venovaná neočakávaným situáciám, než očakávaným.

3. **Vytvorenie spätne volanej funkcie na ošetrenie chýb, ktorú budú pri výskyte výnimky volať nízko-úrovňové funkcie.**

   Tento spôsob používa jazyk C++ napríklad na ošetrenie chýb alokácie dynamickej pamäti. Funkcia **set_new_handler()** umožňuje aplikačným programom nastaviť užívateľskú obslužnú funkciu, ktorú systém zavolá počas chodu programu, keď nie je možné uspokojiť požiadavku na alokáciu pamäti. Tento spôsob sa často používa v spojení s predchádzajúcim spôsobom, ale zároveň dovoľuje programom definovať obslužné podprogramy, ktoré sa prinajmenšom pokúsia obnoviť stav pred chybou.

   Zoberme napríklad spätne volanú funkciu, inštalovanú pomocou **set_new_handler()**. Spätne volaná funkcia by mohla napríklad uložiť blok pamäti na disk a znovu sa pokúsiť o alokáciu pamäti. Ak aj druhá alokácia zlyhá, oznámi sa výnimka, inak program pokračuje normálne ďalej.

4. **Vykonanie vzdialeného skoku.**

   V jazyku C môžeme použiť knižničné funkcie **setjmp()** a **longjmp()** na to, aby nízko-úrovňové funkcie zregenerovali zásobník a aby sa vykonávanie mohlo vrátiť do bodu, označené príkazom **setjmp()**. Funkciou **setjmp()** uložíme známy dobre definovaný stav v programe, a keď sa dostaneme do problémov, volaním funkcie **longjmp()** sa tento uložený stav obnoví. Toto opäť predstavuje vysoký stupeň väzby medzi miestom, kde je stav uložený a miestom, kde sa chyba vyskytne.

5. **Vyslanie signálu.**

   Signál je synchrónne prerušenie, ktoré žiada systém, aby zavolal už skôr inštalovaný obslužný program signálu. Obslužné programy signálov sa podobajú inštalovateľným spätne volaným funkciám, popísaným v bode 3, ale implementácia a použiteľnosť signálov je implementačne závislá a častejšie je to funkcia operačného systému a hardvéru, než programovacieho jazyka.

Ak by sme chceli uvedené schémy ošetrenia chybového stavu aplikovať v jazyku C++, vzniká ďalší kritický problém. Vyššie uvedené spôsoby ošetrenia výnimky nevolajú deštruktory, takže lokálne objekty sa ' 'neupracú' poriadne. Toto prakticky znemožňuje efektívnu regeneráciu z výnimočného stavu, pretože vždy necháme za sebou neupratané objekty, t.j. objekty, ku ktorým už viac nemáme prístup:

```
// setjmp() & longjmp()
#include <iostream>
#include <csetjmp>
using namespace std;

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

jmp_buf jbuf;

void fun()
{
  A a;
  for(int i=0;i<3;i++) {
    cout << "fun()\n";
  }
  longjmp(jbuf,1);
}

int main()
{
  if(setjmp(jbuf) == 0) {
    cout << "setjmp=0...\n";
    fun();
  } else {
    cout << "setjmp!=0..." << endl;
  }
  return 0;
}
```

Zavolanie funkcie **setjmp()** uchová všetky relevantné informácie o aktuálnom stave procesora (napríklad obsah čítača inštrukcií a ukazovateľ zásobníka) v premennej **jbuf** typu **jmp_buf** a vráti nulu. V tomto prípade sa chová ako obyčajná funkcia. Avšak ak zavoláme funkciu **longjmp()** použijúc tú istú premennú **jbuf**, je to ako keby sa opäť vykonal návrat z funkcie **setjmp()** – vynoríme sa za **setjmp()**. V tomto prípade však sa vráti hodnota, ktorá je uvedená ako druhý parameter funkcie **longjmp()**, čím môžeme detekovať, že

sme sa vrátili z **longjmp()**. Rozdiel medzi lokálnym príkazom skoku **goto** (s návestím) a nelokálnym je v tom, že sa môžeme vrátiť na akékoľvek dopredu určené miesto vyššie v aktuálnom zásobníku pomocou volaní **setjmp/longjmp** (všade tam, kde umiestnime volanie **setjmp()**).

Problém v C++ spočíva v tom, že funkcia **longjmp()** nerešpektuje objekty. Konkrétne pri skoku mimo rozsah nevolá deštruktory[1]. Volania deštruktorov sú však veľmi dôležité, a tak takýto postup nebude v C++ fungovať korektne. Norma C++ stanovuje, že vetvenie (prostredníctvom **goto** alebo **longjmp**) mimo rozsah, v ktorom má zásobníkový objekt deštruktor, predstavuje ***nedefinované*** správanie.

OOP prístup k ošetrovaniu výnimiek

Ako vidíme výnimky sa ťažko definujú a ešte ťažšie ošetrujú jednotnou formou. Pre formalizovanie ošetrovania výnimiek boli vytýčené tri podmienky::

- Ošetrenie výnimiek by malo byť vlastnosťou programovacieho jazyka, aby sa zabezpečilo, že ošetrenie výnimiek nebude implementačne závislé, a zároveň aby sa do zmätku metodológií ošetrovania výnimiek zaviedol určitý poriadok.

- Spôsob ošetrovania výnimiek by mal byť OOP orientovaný, aby podporoval dedičnosť a polymorfizmus pri ošetrovaní výnimiek.

- Ošetrovanie výnimiek musí byť flexibilné, musí podporovať čo najviac najbežnejších typov výnimiek a ich ošetrenie. Mechanizmus ošetrovania výnimiek musí byť prekrývateľný programátorom a implicitné spracovanie by nemalo narušovať existujúci kód.

Trvalo niekoľko rokov, kým sa ošetrovanie výnimiek dostalo do ANSI normy C++. Vyžadovalo si to úsilie mnohých ľudí, ale výsledok nie je len atraktívny, ale dokonca i užitočný.

Jednou z hlavných vlastností C++ sa stalo tzv. ***ošetrovanie výnimiek***, čo predstavuje lepší spôsob uvažovania o ošetrení chýb. O ošetrovaní výnimiek môžeme povedať, že:

- Písanie kódu na ošetrenie chýb nie je až tak nudné a tento kód nie je premiešaný s "normálnym" kódom. Najskôr píšeme kód, ktorý máme v úmysle písať, a neskôr v samostatnej sekcii napíšeme kód, ktorý sa bude zaoberať chybami. Ak voláme funkciu viackrát, chybový stav z tejto funkcie ošetríme na jednom mieste.

- Chyby nesmieme ignorovať. Ak funkcia potrebuje poslať chybovú správu volajúcemu, "pošle" objekt, reprezentujúci chybu, mimo funkciu. Ak volajúci chybu nezachytí a nespracuje, prejde do ďalšej obaľujúcej sféry, atď. až kým ***niekto*** chybu nezachytí.

Vysielanie výnimky

C++ spôsob ošetrovania výnimiek je založený na koncepcii vzdialeného skoku. Keď sa detekuje výnimka, t.j. ak v kóde narazíme na výnimočnú situáciu, (na situáciu, v ktorej z aktuálneho kontextu nemáme dostatok informácií, aby sme mohli rozhodnúť čo robiť), môžeme poslať informáciu o chybe do rozsiahlejšieho kontextu vytvorením objektu, ktorý bude obsahovať informácie o výnimočnom stave a "vyslať" tento objekt mimo aktuálny kontext. Hovoríme, že program ***vysiela výnimku***. Vyslanie výnimky spôsobí, že program pred pokračovaním vykonávania vykoná vzdialený skok.

Vypadá to nasledovne:

```
#include <iostream.h>

const int CHYBA = -1;

void Fun(unsigned char *data,long size)
{
```

[1] Niektoré kompilátory C++ majú vylepšenú funkciu **longjmp()** o upratovanie objektov v zásobníku. Takéto správanie je však neprenosné.

```
  if(size >1000)
     // nedokážem spracovať tak veľa dát
     throw CHYBA;
  // ... spracovanie dát
}
```

Keď sa vykoná príkaz vyslania výnimky, funkcia **Fun()** sa ukončí a vykonávanie pokračuje na inom mieste. Príkazy, ktoré nasledujú za príkazom **throw** sa preskočia. V tomto zmysle je príkaz **throw** podobný príkazu **return**, s tým rozdielom, že vykonávanie nepokračuje príkazom, ktorý nasleduje za riadkom, z ktorého sa funkcia volala.

Vysielanie inicializovaných objektov

Vyššie uvedený príklad ilustruje použitie tzv. ***skalárnej výnimky*** v príkaze **throw**. Avšak výnimka môže byť reprezentovaná objektom. Prostredníctvom objektu môžeme do obslužného podprogramu výnimky preniesť doplňujúce informácie:

```
// Výnimkový trieda
class Chyba {
   const char* const Oznam_d;
public:
   Chyba(const char* const oznam = 0) : Oznam_d(oznam) {}
};

void Fun() {
   // Vyslanie výnimkového objektu
   throw Chyba("Delenie nulou!");
}

int main()
{
  // Ako za chvíľu uvidíme, tu budeme potrebovať try blok
  Fun();
  return 0;
}
```

Trieda **Chyba** je obyčajná trieda, ktorá v tomto prípade akceptuje ako parameter konštruktora smerník na pole znakov **char ***. Vysielať môžeme akýkoľvek typ, ale zvyčajne na vysielanie výnimiek vytvárame špeciálne triedy.

Kľúčové slovo **throw** spôsobí, že nastane niekoľko relatívne záhadných udalostí. Po prvé, vytvorí kópiu objektu, ktorý vysielame a "vráti" ho z funkcie, obsahujúcej **throw** výraz, dokonca i v prípade i keď typ objektu nie je to, čo má táto funkcia vracať. Jednoducho si ošetrenie výnimky môžeme predstaviť ako alternatívny mechanizmus návratu (i keď tomu nie je tak). Vyslaním výnimky môžeme odísť i z obyčajného rámca. V každom prípade sa vráti hodnota a odíde sa z rámca funkcie.

Akákoľvek podobnosť s návratom funkcie tu končí, pretože po vyslaní výnimky miesto ***kde*** sa vrátime je celkom odlišné, než ako pri obyčajnom návrate z funkcie. (Skončíme v príslušnej časti kódu – nazývanej ***ošetrenie výnimky*** – ktorá môže byť od miesta, v ktorom bola výnimka vyslaná, veľmi ďaleko). Okrem toho sa zrušia všetky lokálne objekty, vytvorené do výskytu výnimky. (Normálny návrat z funkcie predpokladá, že zrušené musia byť všetky objekty v rámci, vrátane tých, ktoré boli vytvorené za miestom výskytu výnimky). Toto automatické upratovanie lokálnych objektov sa často nazýva ***odvíjanie zásobníka***. Samozrejme samotný výnimkový objekt sa na zodpovedajúcom mieste tiež korektne uprace.

Vysielať môžeme toľko rôznych typov objektov, koľko chceme. Zvyčajne vysielame odlišný typ pre každú kategóriu chyby.

Zachytávanie výnimky

Ak funkcia vysiela výnimku, musí predpokladať, že výnimka sa zachytí, a že niekto sa bude ňou zaoberať. Ako sme už spomenuli, jednou z výhod ošetrovania výnimky v C++ je, že dovoľuje koncentrovať sa na problém, ktorý sa vlastne snažíme riešiť na jednom mieste, a potom sa zaoberať chybami z tohto kódu na inom mieste.

Try blok

Ak sme vo vnútri funkcie a vyšleme výnimku (alebo volaná funkcia vyšle výnimku), funkcia sa skončí procesom vyslania. Funkcia, ktorá chce ošetrovať výnimky, musí byť vložená do špeciálneho bloku, nazývaného ***try*** blok. Try blok je obyčajný rámec, ktorý začína kľúčovým slovom **try**.

```
try {
  // Kód, ktorý môže generovať výnimky
}
```

Ak kontrolujeme chyby pozorným testovaním návratových kódov z funkcie, musíme každé volanie funkcie obaliť nastavovacím a testovacím kódom, i keď tú istú funkciu voláme niekoľkokrát. Ak použijeme ošetrovanie výnimky, všetko vložíme do **try** bloku bez kontroly chýb. Kód sa bude ľahšie písať i čítať, pretože kód nie je poprepletaný testovaním chybového stavu.

Rutina ošetrenie výnimky

Samozrejme vyslanie výnimky musí niekde skončiť. Po vyslaní výnimky za chodu programu sa vykonávanie prenesie do najbližšieho tzv. ***catch*** bloku, ktorý dokáže ošetriť vygenerovanú výnimku daného typu. Toto miesto sa nazýva ***rutina ošetrenia výnimky***. Pre každý typ výnimky, ktorú chceme zachytiť potrebujeme jednu rutinu. Rutiny ošetrenia výnimiek nasledujú ihneď za **try** blokom a začínajú kľúčovým slovom **catch**:

```
try {
  // Kód, ktorý môže generovať výnimky
}
catch(typ1 id1) {
  // Ošetrenie výnimky typu typ1
}
catch(typ2 id2) {
  // Ošetrenie výnimky typu typ2
}
catch(typ3 id3)
  // atď...
}
catch(typN idN)
  // Ošetrenie výnimky typu typN
}
// Tu pokračuje normálne vykonávanie programu...
```

Každý **catch** blok (rutina ošetrenia výnimky) vypadá ako malá funkcia, ktorá akceptuje jeden argument jedného konkrétneho typu. Identifikátor (**id1, id2** atď.) môžeme vo vnútri rutiny používať podobne ako argument funkcie, ale tento identifikátor môžeme i vynechať, ak ho rutina nepotrebuje. Už typ výnimky zvyčajne poskytuje dostatočné informácie o čo sa jedná.

Rutiny musia byť umiestnené priamo za **try** blokom. Po vyslaní výnimky mechanizmus ošetrenia výnimky hľadá prvú rutinu s takým typom argumentu, ktorý sa zhoduje s typom výnimky. Ak ju nájde, vojde do vnútra tohto **catch** bloku a výnimka sa považuje za ošetrenú. (Po nájdení **catch** bloku sa hľadanie zodpovedajúceho bloku zastaví). Vykoná sa iba tento zodpovedajúci **catch** blok a riadenie sa vráti za posledný **catch** blok, patriaci tomuto **try** bloku.

V rámci try bloku môže generovať ten istý typ výnimky viac rôznych funkcií, ale na ošetrenie potrebujeme iba jednu rutinu.

V podstate počet výnimiek, ktoré možno ošetriť jedným typom výnimky nie je ohraničený, ale použitie jednoduchého typu na zachytenie viac ako jednej výnimky vedie k použitiu príkazov **if** alebo **switch**. Lepším spôsobom je použitie rôznych typov (presne povedané tried) pre každú výnimku a rozčlenenie detailov ponecháme na dedičnosti a virtuálnych funkciách. ale o tom neskôr.

A na ilustráciu použitia **try** a **catch** nahradíme **setjmp** try blokom a volanie **longjmp** nahradíme príkazom **throw**.

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

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

void Fun()
{
  A a;
  for(int i = 0; i < 3; i++)
    cout << "fun()\n";
  throw 47;
}

int main() {
  try {
    cout << "main()" << endl;
    Fun();
  }
  catch (int) {
    cout << "catch(int) "<< endl;
  }
}
```

Keď sa vo funkcii **Fun()** vykoná príkaz **throw**, riadenie programu sa vracia (ustupuje), až kým sa nenájde **catch** blok, ktorý akceptuje parameter typu **int**, a vykonávanie sa obnoví vykonávaním tela tohto **catch** bloku. Najdôležitejším rozdielom medzi týmto programom a programom so **setjmp/longjmp** je, že pre objekt **a** sa zavolá deštruktor, keď príkaz **throw** spôsobí, že vykonávanie opustí funkciu **Fun()**.

Ukončenie a obnovenie

V teórií ošetrovania výnimiek existujú dva základné modely: ***ukončenie a obnovenie***. V ukončení (čo podporuje i C++) predpokladáme, že chyba je tak kritická, že sa nám v žiadnom prípade nepodarí automaticky obnoviť vykonávanie v bode, kde výnimka nastala. Inými slovami "ktokoľvek" vyslal výnimku, rozhodol sa, že situácia sa nedá zachrániť a ***nechceme*** sa vrátiť späť.

Alternatívny model ošetrovania chýb sa nazýva ***obnovenie***. Použite obnovovacej sémantiky znamená, že od rutiny ošetrenia výnimky sa očakáva, že urobí niečo, čo napraví situáciu a potom sa automaticky zopakuje poruchový kód, predpokladajúc, že na druhý raz sa jeho vykonanie úspešne podarí. Ak chceme obnovovanie v C++ aplikovať, musíme explicitne preniesť vykonávanie späť do kódu, kde chyba nastala, zvyčajne opakovaním volania funkcie, ktorá nás tam poslala. Nie je to nič nezvyčajné, pretože umiestnenie try bloku do vnútra slučky **while** nás udrží v **try** bloku až kým nie je výsledok uspokojivý.

Hoci obnovovanie znie na prvé počutie atraktívne, zdá sa, že nie v praxi až také užitočné. Jedným z dôvodov môže byť i vzdialenosť medzi výskytom výnimky a jej obslužnou rutinou. Jedná vec je skočiť do rutiny a skončiť, ale skočiť do rutiny a potom späť môže byť pre rozsiahle systémy koncepčne ťažké, najmä v prípadoch kde môžu byť výnimky generované z mnohých miest.

Vyhľadanie správnej obsluhy výnimky

Príkazy **throw** sú dokonalejšie a bezpečnejšie ako **longjmp()**. V príkaze **throw** nešpecifikujeme miesto kam sa má skočiť, ale iba druh výnimky, ktorá nastala. Kde bude program pokračovať musí určiť kompilátor. Keď kompilátor nájde príkaz **throw**, generuje kód, ktorý počas chodu prehľadá reťaz volaní funkcií a nájde prvú funkciu, obsahujúcu príkaz **catch**, ktorý je schopný ošetriť vyslanú výnimku. Ak sa vhodný príkaz **catch** nenájde, použije sa implicitné ošetrenie výnimky, ktoré programu ukončí.

Rozbalenie zásobníka

Pri hľadaní správnej obsluhy výnimky kompilátor volá kód, ktorý určí ktorá funkcia v reťazi volaní má vyhovujúci príkaz **catch**. Pri stúpaní cez zásobník pri hľadaní **catch** príkazu kompilátor volá deštruktory všetkých lokálnych objektov, ktorých zásobníkové obrazy zostali v zásobníku. Kompilátor zabezpečí, že deštruktory odvodených tried sa volajú pred deštruktormi základných tried, kopírujúc tak normálnu postupnosť volaní deštruktorov, ktorá sa vykonáva pri rušení objektu. Pre objekty, obsahujúce subobjekty kompilátor zabezpečí, že deštruktory sa zavolajú iba pre objekty, ktorých konštruktory sa vykonali celé.

Ak ošetrovací kód nemôže nájsť vhodnú obsluhu vygenerovanej výnimky, zásobník sa celkom odbalí a zavolá sa funkcia **terminate()**, ktorá volá funkciu **abort()**, ale dá sa inštalovať vlastná rutina, ktorú bude funkcia **terminate()** volať.

Porovnávanie výnimiek

Po vyslaní výnimky systém ošetrenia výnimiek prechádza cez "najbližšie" obslužné rutiny v poradí, v akom sú uvedené v zdrojovom kóde. Keď nájde zhodu, výnimka sa považuje za spracovanú a ďalej sa už nehľadá.

Vyhľadávanie zodpovedajúcej výnimky nevyžaduje absolútnu koreláciu medzi výnimkou a jej obslužnou rutinou. Objekt alebo odkaz na odvodený objekt nájde obslužnú rutinu, zachytávajúcu základnú triedu (avšak ak obslužná rutina zachytáva objekt a nie odkaz, objekt výnimky sa pri prenose do obslužnej rutiny oreže na základný typ. Toto nespôsobí haváriu, ale stratia sa všetky informácie z odvodenej triedy.) Aby sme sa vyhli vytváraniu kópie objektu výnimky, je vždy lepšie zachytávať výnimky odkazom a nie hodnotou. Ak vyšleme smerník, pri hľadaní zhody sú použité zvyčajné štandardné konverzie smerníkov. Avšak pri hľadaní zhody sa nepoužívajú žiadne automatické konverzie typov z jednej typu výnimky na iný typ, napríklad:

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

class Vynimka1 {
};

class Vynimka2 {
   public:
     Vynimka2(const Vynimka1&) {}
   };

void Fun()
{
  throw Vynimka1();
}

int main() {
```

```
  try {
    Fun();
  }
  catch (Vynimka2&) {
    cout << "catch(Vynimka2)" << endl;
  }
  catch (Vynimka1&) {
    cout << " catch(Vynimka1)" << endl;
  }
}
```

Na prvý pohľad sa zdá, že by sa mala použiť prvá obslužná rutina konverziou objektu triedy **Vynimka1** na objekt triedy **Vynimka2**, použijúc konverzný konštruktor. Systém takúto konverziu počas ošetrenia výnimky neurobí a skončíme v obslužnej rutine pre výnimku **Vynimka1**.

Nasledujúci kód ilustruje ako obslužná rutina základnej triedy dokáže zachytávať výnimky odvodených tried:

```
// Hierarchia výnimiek
#include <iostream>
using namespace std;

class Chyba {
};

class Varovanie : public Chyba {
};

class Fatal : public Chyba {
};

class A {
  public:
    void Fun() { throw Fatal(); }
};

int main() {
  A a;
  try {
    a. Fun ();
  }
  catch(Chyba &) {
    cout << "Chyba" << endl;
  }
  // Skryté predchádzujúcim blokom
  catch(Varovanie &) {
    cout << "Varovanie" << endl;
  }
  catch(Fatal &) {
    cout << "Fatal" << endl;
  }
}
```

V tomto príklade mechanizmus ošetrenia výnimiek vždy nájde a vykoná catch blok, zachytávajúci objekt **Chyba**. V tomto bloku sa zastaví všetko *čo je* **Chyba** (prostredníctvom **public** dedičnosti) v prvej obslužnej rutine. Znamená to, že druhá a tretia rutina sa nikdy nezavolajú, pretože všetko zachytí prvá. Je lepšie zachytávať najskôr odvodené typy a základný typ dávať až na koniec, aby sa zachytili menej špecifické výnimky.

Všimnime si, že uvedené príklady zachytávajú výnimky odkazom, hoci pre tieto triedy to nie je dôležité, pretože odvodené triedy neobsahujú žiadne ďalšie členy a obslužné rutiny nemajú žiaden identifikátor argumentu. Zvyčajne v obslužných rutinách používame odkazové argumenty namiesto hodnotových, aby nedochádzalo k orezávaniu informácie.

Zachytávanie akejkoľvek výnimky

Občas potrebujeme vytvoriť obslužnú rutinu, ktorá bude ***zachytávať akýkoľvek*** typ výnimky. Toto robíme pomocou tzv. vynechávky (...) v zozname argumentov:

```
catch(...) {
  cout << "Zachytená neidentifikovateľná výnimka" << endl;
}
```

Značka vynechávky zachytáva akúkoľvek výnimku, takže by sme ju mali dávať až na koniec zoznamu obslužných rutín, aby sme zabránili predčasnej obsluhe výnimky.

Pretože vynechávky nedávajú žiadnu možnosť použiť argument, o výnimke nevieme vôbec nič, dokonca nepoznáme ani jej typ. Táto rutina funguje ako "zachytávač".

Znovuvyslanie výnimky

Vyslať tú istú výnimku v rámci ošetrenia výnimky zvyčajne potrebujeme, keď máme alokovaný nejaký prostriedok, napríklad sieťové spojenie alebo dynamickú pamäť z haldy, ktorý musíme dealokovať. Ak nastane výnimka, nezaujíma nás čo v aktuálnom kontexte chybu spôsobilo, akurát potrebujeme napríklad ukončiť pripojenie, ktoré sme predtým otvorili. Po tomto chceme, aby nejaký ďalší kontext, bližší k užívateľovi (t.j. vyššie v reťazi volaní) výnimku spracoval. V takomto prípade je vynechávka presne to, čo potrebujeme. Potrebujeme zachytiť akúkoľvek výnimku, upratať naše prostriedky, a potom výnimku znovu vyslať tak, aby sa ošetrila niekde inde. Výnimku znovu vyšleme použitím kľúčového slova **throw** bez argumentu vo vnútri obslužnej rutiny výnimky:

```
catch(...) {
  cout << "Zachytená výnimka" << endl;
  // Dealokácia prostriedkov a znovuvyslanie…
  throw;
}
```

Všetky ďalšie **catch** bloky tohto **try** bloku sa ignorujú. Použitie **throw** spôsobí, že výnimka pôjde do ošetrovacích rutín na vyššej kontextovej úrovni. Okrem toho všetko o objekte výnimky sa zakonzervuje, takže obslužná rutina na vyššej úrovni, ktorá zachytáva špecifický typ výnimky, dokáže extrahovať informácie, ktoré môže výnimkový objekt obsahovať.

Nezachytené výnimky

Ošetrovanie výnimiek sa považuje za lepší spôsob ošetrovania chybového stavu, než tradičná metóda ukončenia pri chybe, pretože výnimky sa môžu ignorovať. Ak žiaden z **catch** blokov, ktoré nasledujú za **try** blokom nevyhovuje výnimke, výnimka sa posunie do kontextu na vyššej úrovni, t.j. do funkcie alebo **try** bloku, ohraničujúcom **try** blok, ktorý výnimku nezachytil (umiestnenie tohto **try** bloku nie je vždy na prvý pohľad zrejmé, pretože je vyššie v reťazci volaní). Tento proces pokračuje, až kým na sa nejakej úrovni obslužná rutina nezhoduje s výnimkou. V tomto okamihu sa výnimka považuje za "zachytenú" a ďalej sa neprehľadáva.

Funkcia terminate

Ak žiadna rutina na žiadnej úrovni výnimku nezachytí, automaticky sa zavolá špeciálna knižničná funkcia **terminate()**, deklarovaná v hlavičkovom súbore **exception**. Implicitne funkcia **terminate()** zavolá

štandardnú knižničnú funkciu **abort()**[2], ktorá program ihneď preruší. Keď sa zavolá funkcia **abort()**, nevykonáva sa žiadna normálna ukončovacia funkcia programu, čo znamená, že sa nevykonajú deštruktory globálnych a statických objektov. Nezachytenú výnimku môžeme považovať za programovaciu chybu. Funkcia **terminate()** sa vykonáva i v prípade, že deštruktor lokálneho objektu vyšle výnimku počas rozbaľovania zásobníka (prerušenie spracovávanej výnimky) alebo ak výnimku vyšle **konštruktor alebo deštruktor globálneho alebo statického objektu**. Všeobecne platí pravidlo, že deštruktor **nesmie** vysielať výnimku.

Funkcia set_terminate

Pomocou štandardnej funkcie **set_terminate()** môžeme inštalovať svoju vlastnú ukončovaciu **terminate()** funkciu. Funkcia **set_terminate()** vracia smerník na **terminate** funkciu, ktorú nahradzujeme (ktorou pri prvom volaní bude štandardná knižničná verzia funkcie), takže v prípade potreby ju môžeme vrátiť do pôvodného stavu. Užívateľská funkcia **terminate()** nesmie mať žiadne argumenty a jej návratová hodnota musí byť **void**. Akákoľvek obslužná ukončovacia rutina nesmie vracať alebo vysielať výnimku. Jediné čo môže, je vykonávanie nejakej logiky, ukončujúcej program. Ak sa zavolá funkcia **terminate()**, problém je neodstrániteľný.

Nasledujúci príklad ilustruje použitie funkcie `set_terminate`. Návratová hodnota volanie tejto funkcie sa uchováva a obnovuje, takže funkcia `terminate` sa dá použiť na izolovanie časti kódu, v ktorom neočakávaná výnimka nastala:

```
// Použitie set_terminate()
// Nezachytené výnimky
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

void terminator()
{
  cout << "terminator()" << endl;
  exit(0);
}

void (*old_terminate)() = set_terminate(terminator);

class A {
  public:
    class AA {
    };
    void Fun() {
      cout << "A::Fun()" << endl;
      throw AA();
    }
    ~A() {
      cout << "~A()" << endl;
      throw 'c';
    }
};

int main()
{
  try {
    A a;
    a.Fun();
  }
  catch(...) {
    cout << "catch(...)" << endl;
```

[2] V Unix systémoch funkcia **abort** vytvára výpis pamäte (core dump).

```
    }
    return 0;
  }
```

Definícia premennej **old_terminate** vypadá na prvý pohľad trochu zvláštne: nielen že sa vytvára smerník na funkciu, ale zároveň sa smerník i inicializuje na návratovú hodnotu volania funkcie **set_terminate()**. I keď za deklaráciou smerníka na funkciu častejšie vídame bodkočiarku, je to len ďalší druh premennej, ktorú môžeme inicializovať pri definícii.

Trieda **A** vysiela výnimku vo vnútri funkcie **Fun()**, ale tiež i vo svojom deštruktore. Takáto situácia spôsobí volanie funkcie **terminate()**. Dokonca i keď obslužná rutina výnimky zachytáva všetky výnimky - **catch(...)**, a zdá sa, že sa zachytáva všetko a tým pádom nie je dôvod na volanie funkcie **terminate()**, v každom prípade sa táto funkcia zavolá. Počas upratovania objektov v zásobníku pri ošetrovaní jednej výnimky sa zavolá deštruktor objektu triedy **A,** a tento vysiela ďalšiu výnimku, ktorá spôsobí volanie funkcie **terminate()**. Deštruktor, ktorý vysiela výnimku alebo spôsobuje jej vyslanie je chybou návrhu.

Upratovanie

Súčasťou mágie ošetrovania výnimiek je, že môžeme vyskočiť z normálneho toku programu do zodpovedajúcej obslužnej rutiny výnimky. Avšak vykonanie tohto by nebolo užitočné, keby sa pri vyslaní výnimky všetky veci náležíte neupratali. Systém ošetrenia výnimiek v C++ zabezpečuje, že pri odchode z rámca sa pre všetky objekty, ktorých konštruktory boli ukončené, zavolajú deštruktory.

Nasledujúci príklad demonštruje, že pre objekty, ktorých konštruktory sa nevykonali celé, sa nezavolajú príslušné deštruktory. Zároveň ilustruje, čo sa stane, keď sa výnimka vyšle uprostred vytvárania poľa objektov:

```
// Upratovanie pri výnimke uprace iba kompletné objekty
#include <iostream>
using namespace std;

class A {
  private:
    static int Citac_d;
    int ObjId_d;
  public:
    A() {
      ObjId_d = Citac_d++;
      cout << "Konstrukcia objektu triedy A c." << ObjId_d << endl;
      if(ObjId_d == 4)
        throw 4;
    }
   ~A() {
      cout << "Destrukcia objektu triedy A c." << ObjId_d << endl;
  }
};

int A::Citac_d = 0;

int main() {
  try {
    A a;
    // Vyslanie výnimky:
    A pole[5];
    A ax;  // tu sa nedostaneme
  }
  catch(int i) {
    cout << "Zachytene " << i << endl;
```

```
    }
}
```

Trieda **A** počíta svoje objekty, takže môžeme sledovať postup programu. Počet vytvorených objektov sa uchováva v statickom dátovom člene **Citac_d** a číslo konkrétneho objektu v dátovom člene **ObjId_d**.

Hlavný program vytvára jeden objekt **a** triedy **A** (ObjId_d=0) a potom sa pokúša vytvoriť pole piatich objektov triedy **A**, ale pred úplným vytvorením tretieho objektu sa vysiela výnimka. Výsledok programu je nasledovný:

```
Konstrukcia objektu triedy A c.0
Konstrukcia objektu triedy A c.1
Konstrukcia objektu triedy A c.2
Konstrukcia objektu triedy A c.3
Destrukcia objektu triedy A c.2
Destrukcia objektu triedy A c.1
Destrukcia objektu triedy A c.0
Zachytene 3
```

Tri prvky poľa sa úspešne vytvoria, ale v strede konštruktora štvrtého prvku sa vysiela výnimka. Pretože konštrukcia štvrtého prvku sa vo funkcii **main()** (pre pole[5]) nikdy neukončí, zavolajú sa iba deštruktory objektov s **ObjId_d** 1 a 2. Nakoniec sa zruší objekt **a**, ale nie objekt **ax**, pretože sa nikdy nevytvoril.

Správa prostriedkov

Keď píšeme kód s výnimkami, je obzvlášť dôležité, aby sme sa neustále pýtali *"Ak výnimka nastane, budú moje prostriedky správne upratané?"* Väčšinou je to bezpečné, avšak v prípade konštruktorov nastáva problém: Ak sa výnimka vyšle skôr, ako je konštruktor celý vykonaný, pre objekt sa zodpovedajúci deštruktor nezavolá. Z toho vyplýva, že pri písaní konštruktora musíme byť veľmi pozorní.

Všeobecným problémom je alokácia prostriedkov v konštruktoroch. Ak nastane výnimka v konštruktore, deštruktor nedostane šancu prostriedok dealokovať. Problém sa najčastejšie vyskytuje pri tzv. "nechránených" smerníkoch. Napríklad:

```
// Nechránené smerníky
#include <iostream>
using namespace std;

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

class B {
   public:
     void* operator new(size_t sz) {
       cout << "Alokacia B" << endl;
       throw 47;
     }
     void operator delete(void* p) {
       cout << "Dealokacia B" << endl;
       ::operator delete(p);
     }
};

class PouziAaB {
    A* bp;
    B* op;
```

```
public:
  PouziAaB(int pocet = 1) {
    cout << "PouziAaB()" << endl;
    bp = new A[pocet];
    op = new B;
  }
  ~PouziAaB() {
    cout << "~PouziAaB()" << endl;
    delete []bp; // zmazanie pola
    delete op;
  }
};

int main() {
  try {
    PouziAaB ur(3);
  }
  catch(int) {
    cout << "Obsluha vynimky" << endl;
  }
}
```

Výstup je nasledovný:

```
PouziAaB()
A()
A()
A()
Alokacia B
Obsluha vynimky
```

Najskôr sa začne sa vykonávať konštruktor triedy **PouziAaB()**, úspešne sa vykoná konštruktor triedy **A** pre tri objekty poľa. Avšak vo vnútri operátorovej funkcie **B::operator new** sa vysiela výnimka (simulujúca chybu nedostatku pamäti). Náhle sme skončili vo vnútri rutiny bez zavolania deštruktora triedy **PouziAaB**. Je to správne, pretože konštruktor triedy **PouziAaB** sa nemohol skončiť, avšak znamená to, že objekty triedy **A** už boli úspešne v halde alokované, ale nikdy sa nezrušia.

Všetko ako objekt

Aby sme zabránili takýmto únikom prostriedkov, musíme sa pred takýmito "surovými" alokáciami prostriedkov brániť. K dispozícií sú dve možnosti:

- Výnimku zachytíme vo vnútri konštruktora a alokované prostriedky uvoľníme
- Alokácie vložíme do konštruktora objektu a dealokácie dáme do deštruktora objektu – tzv. ***obalové triedy***.

Použitím druhého prístupu sa každá alokácia stáva atomickou, pretože sa stáva súčasťou doby života lokálneho objektu, a ak zhavaruje, ostatné objekty, alokujúce prostriedky sa korektne upracú počas odvíjania zásobníka.

Obalové triedy sú v C++ veľmi dôležité, ak používame výnimky. Všetky zdroje by mali byť obalené, aby sme zabránili napr. strate pamäti, atď. Zoberme nasledujúcu situáciu:

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

class A {
    int *Pole_d;
  public:
```

```
    A() {
      cout << "A()" << endl;
      Pole_d=new int[100];
      rob_nieco();
    };
    void rob_nieco() {
      Pole_d[0]=1;
      throw 47;
    };
   ~A() {
      cout << "~A()" << endl;
      delete []Pole_d;
    };
};


int main() {
  try {
    A ur;
  }
  catch(int) {
    cout << "Obsluha vynimky" << endl;
  }
}
```

Výstup je nasledovný:

```
A()
Obsluha vynimky
```

Kód vypadá správne, až kým nezačneme uvažovať nad tým, čo sa stane, ak sa vo vnútri funkcie **A::rob_nieco()** vygeneruje výnimka. Pretože funkcia sa volá vo vnútri konštruktora, objekt **A** ešte nebol úplne vytvorený. Kód skočí do najbližšieho kompatibilného ošetrenia výnimky, čím sa preskočí deštruktor triedy **A**. Kód vytvorí v pamäti trhlinu, pretože sa nezavolá deštruktor triedy **A**, a tým sa nedealokuje pole **Pole_d**.

Jedným z riešení by bolo umiestniť volanie funkcie **rob_nieco()** pred alokáciu pamäti. Zvyčajne to však nie je možné pretože funkcia **rob_nieco()** používa alokovanú pamäť. Naviac presne poradie kódu by nemalo určovať, či vznikne v pamäti diera alebo nie. Úplné riešenie spočíva v obalení pamäťového prostriedku do obalovej triedy a celý program prepísať nasledovne:

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

#include <iostream>
using namespace std;

class Obal {
    int *Pole_d;
  public:
    Obal(int vel) {
      cout << "Obal() - alokujem 'vel' int" << endl;
      Pole_d=new int[vel];
    };
   ~Obal() {
      cout << "~Obal() - dealokujem 'vel' int" << endl;
      delete [] Pole_d;
    };
      operator int *() { return Pole_d; };
};
```

```
class A {
    Obal Pole_d;
  public:
    A() : Pole_d(100) {
      cout << "A()" << endl;
      rob_nieco();
    };
    void rob_nieco() {
      Pole_d[0]=1;
      throw 47;
    };
   ~A() {
      cout << "~A()" << endl;
    };
};


int main() {
  try {
    A ur;
  }
  catch(int) {
    cout << "Obsluha vynimky" << endl;
  }
}
```

Výstup je nasledovný:

```
Obal() - alokujem 'vel' int
A()
~Obal() - dealokujem 'vel' int
Obsluha vynimky
```

Vidíme, že sa alokované pole korektne dealokuje i v prípade, že sa konštruktor triedy **A** nevykoná celý.

Takúto triedu môžeme včleniť triedu všade tam, kde ju potrebujeme so zárukou, že kompilátor bude vždy dealokovať alokovanú pamäť a to i v prípade výnimky. Pozrime sa ako by mohlo vypadať použitie triedy **Obal** vo funkcii:

```
void Fun()
{
  Obal tab(100);

  // pouzi objekt tab
  //...
}
```

Na upratanie objektov triedy **Obal** nie je potrebné ošetrenie výnimky. Samozrejme nič nám nebráni zachytávať výnimky, spôsobené inými dôvodmi, ale náš kód prinajmenšom zaručuje, že kompilátor nebude vytvárať diery v pamäti.

Try bloky na úrovni funkcie

Pretože konštruktory môžu bežne vysielať výnimky, mohli by sme chcieť ošetrovať výnimky, ktoré nastanú pri inicializácií objektového člena alebo inicializácií subobjektu. Aby sme to mohli urobiť, musíme inicializáciu takýchto subobjektov umiestniť do ***try bloku na úrovni funkcie***. Na rozdiel od obyčajnej syntaxe, try blokom pre konštruktorové inicializátory je telo konštruktora a zodpovedajúci **catch** blok nasleduje za telom konštruktora:

```
// Ošetrenie výnimiek v subojektoch
#include <iostream>
using namespace std;

class cZakladnaTrieda {
     int i;
   public:
     class cVynimka {};

     cZakladnaTrieda(int i) : i(i) {
       throw cVynimka();
     }
};

class cOdvodenaTrieda : public cZakladnaTrieda {
   public:
     class cVynimkaOdv {
       const char* msg;
     public:
       cVynimkaOdv(const char* msg) : msg(msg) {}
       const char* Oznam() const {
         return msg;
       }
     };
     cOdvodenaTrieda(int j) // Konštruktor
     try
       : cZakladnaTrieda(j) {
       // Telo konštruktora
       cout << "Toto by sa nemalo vytlačit" << endl;
     }
     catch (cVynimka &) {
       throw cVynimkaOdv("cZakladnaTrieda subobjekt vyslal vynimku");;
     }
};

int main() {
  try {
    cOdvodenaTrieda d(3);
  }
  catch (cOdvodenaTrieda::cVynimkaOdv& d) {
    cout << d.Oznam() << endl;
          // cZakladnaTrieda subobjekt vyslal vynimku"
  }
}
```

Všimnime si, že inicializačný zoznam v konštruktore triedy **cOdvodenaTrieda** je až za kľúčovým slovom **try**, ale pred telom konštruktora. Ak nastane výnimka, obsiahnutý objekt sa nevytvorí, takže nemá zmysel vracať sa ku kódu, ktorý ho vytváral. Z tohto dôvodu jediným rozumným riešením je vyslanie výnimky v **catch** klauzule na úrovni funkcie.

Hoci toto nie je až tak používané, C++ dovoľuje try bloky na funkčnej úrovni pre akúkoľvek funkciu:

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

int main() try {
  throw "main";
}
```

```
catch(const char* msg) {
  cout << msg << endl;
  return 1;
}
```

V tomto prípade používa **catch** blok na návrat príkaz **return** rovnakým spôsobom ako telo funkcie. Použitie tohto typu try bloku na funkčnej úrovni sa moc neodlišuje od vloženia **try-catch** okolo kódu vo vnútri tela funkcie.

Štandardné výnimky

Štandardná C++ knižnica obsahuje množinu výnimiek, ktoré môžeme používať pri programovaní. Je ľahšie a rýchlejšie začínať s triedami štandardných výnimiek, než definovať svoje vlastné. Ak štandardná trieda presne nespĺňa požiadavky, môžeme z nej vytvárať potomkov použitím dedenia.

Všetky triedy štandardných výnimiek sa odvádzajú z triedy **exception**, ktorá je definovaná v hlavičkovom súbore **<exception>**. Dve odvodené triedy sa nazývajú **logic_error** a **runtime_error**, ktoré nájdeme v hlavičkovom súbore **<stdexcept>**. Trieda **logic_error** reprezentuje chyby programovacej logiky, napríklad prenos nesprávneho argumentu. Chyby chodu programu sú také, ktoré nastávajú ako výsledok nepredvídaného vplyvu ako je chyba hardvéru alebo vyčerpanie pamäti. Obidve triedy **runtime_error** i **logic_error** obsahujú konštruktor, ktorý akceptuje parameter typu **std::string**, takže dokážu uchovávať v objekte výnimky oznam, ktorý neskôr môžeme získať metódou **exception::Oznam**:

```
// Odvodenie výnimkovej triedy od std::runtime_error
#include <stdexcept>
#include <iostream>
using namespace std;

class Chyba : public runtime_error {
   public:
     Chyba(const string& msg = "") : runtime_error(msg) {}
};

int main() {
  try {
    throw Chyba("Moj oznam");
  }
  catch (Chyba& x) {
    cout << x.Oznam() << endl;
  }
}
```

Hoci konštruktor triedy **runtime_error** posiela oznam do subobjektu **std::exception**, trieda **std::exception** neposkytuje konštruktor, ktorý akceptuje argument typu **std::string** a preto zvyčajne výnimkové triedy odvodzujeme z tried **runtime_error** alebo **logic_error** (alebo z ich potomkov a nie zo **std::exception**).

Triedy štandardných výnimiek sú popísané v nasledujúcich tabuľkách:

exception	Základná trieda pre všetky výnimky, vysielané C++ štandardnou knižnicou. Na získanie voliteľného reťazca, ktorý bol použitý pri inicializácií výnimky, môžeme použiť metódu **what**.
logic_error	Odvodená z triedy **exception**. Oznamuje chyby logiky programu, ktoré sa dajú zistiť testovaním.
runtime_error	Odvodená z triedy **exception**. Oznamuje chyby chodu programu, ktoré sa dajú zistiť len za chodu programu.

Z triedy **exception** je tiež odvodená trieda výnimky **ios::failure**, ale nemá už ďalšie podtriedy.

Uvedené triedy môžeme používať tak ako sú uvedené v tabuľkách alebo ich môžeme použiť ako základné triedy, z ktorých odvodíme svoje vlastné špecifickejšie typy výnimiek.

Triedy výnimiek, odvodené z triedy **logic_error**

domain_error	Oznamuje porušenie predbežnej podmienky.
invalid_argument	Indikuje neplatný argument funkcie, z ktorej je vyslaná.
length_error	Indikuje pokus vytvoriť objekt, ktorého dĺžka je väčšia alebo rovná hodnote **npos** (najväčšia reprezentovateľná hodnota typu **size_t**).
out_of_range	Oznamuje argument mimo medze.
bad_cast	Vysielaná pri vykonaní neplatného výrazu **dynamic_cast** pri identifikácií typu za chodu programu.
bad_typeid	Oznamuje null smerník **p** vo výraze **typeid(*p)**. (Týka sa identifikácie typu za chodu programu).

Triedy výnimiek, odvodené z triedy **runtime_error**

range_error	Oznamuje narušenie post-podmienky.
overflow_error	Oznamuje aritmetické pretečenie.
bad_alloc	Oznamuje chybu alokácie pamäti.

Špecifikácia výnimky

Nie je nutné, aby sme užívateľov našich funkcií informovali, aké výnimky naše funkcie môžu vysielať. Avšak ak na to zabudneme, znamená to, že užívatelia si nemôžu byť istí, aký kód majú písať, aby zachytili všetky potencionálne výnimky. Samozrejme ak je k dispozícií zdrojový kód, môžu hľadať príkazy **throw**, ale knižnica sa zvyčajne nedodáva so zdrojovým kódom. Tento problém môže zmierniť dobrá dokumentácia, ale koľko softvérových projektov je dobre zdokumentovaných? C++ poskytuje syntax, ktorá dovoľuje užívateľovi oznámiť, aké výnimky funkcia vysiela, takže užívateľ ich dokáže ošetriť. Toto sa nazýva ***špecifikácia výnimky***, ktorá je súčasťou deklarácie funkcie a zadáva sa za zoznamom argumentov.

Špecifikácia výnimky tiež používa kľúčové slovo **throw**, za ktorým (v okrúhlych zátvorkách) nasleduje zoznam všetkých typov potencionálnych výnimiek, ktoré môže funkcia vysielať. Deklarácia funkcie by mohla vypadať nasledovne:

```
void f() throw(int, cMojaVynimka, exception);
```

Z pohľadu výnimiek tradičná deklarácia funkcie

```
void f();
```

znamená, že funkcia môže vysielať ľubovoľný typ výnimiek. Ak napíšeme

```
void f() throw();
```

funkcia nebude vysielať žiadne výnimky (a preto je potrebné ubezpečiť sa, že žiadna funkcia ďalej v reťazi volaní nevysiela žiadnu výnimku!).

Kvôli dobrej programovacej stratégii, dobrej dokumentácií a jednoduchosť používania je nutné pri písaní funkcií, ktoré vysielajú výnimky (až na niektoré výnimky - pozri ďalej) vždy poriadne zvážiť použitie špecifikácie výnimiek.

Funkcia unexpected

Čo sa stane ak špecifikácia výnimiek funkcie tvrdí, že sa chystáme vysielať určitú množinu výnimiek a potom z funkcie vyšleme niečo iné, než množina obsahuje? Aký je trest? Nuž ak vyšleme niečo iné, než sa objaví v špecifikácií výnimiek, zavolá sa funkcia **unexpected()**. Ak takáto neočakávaná situácia nastane, implicitná implementácia funkcie **unexpected()** vola funkciu **terminate()**, o ktorej sme už hovorili.

Funkcia set_unexpected

Podobne ako pre funkciu **terminate()** i mechanizmus funkcie **unexpected()** dovoľuje inštalovať svoju vlastnú funkciu ako odozvu na neočakávané výnimky. Túto funkciu nastavujeme volaním funkcie **set_unexpected()**, ktorej parametrom, podobne ako vo funkcii **set_terminate()**, je adresa funkcie bez parametrov a s návratovou hodnotou typu **void**. Podobne, pretože tiež vracia hodnotu smerníka predchádzajúcej funkcie **unexpected()**, môžeme si tento smerník odložiť a neskôr vrátiť systém do pôvodného stavu. Ak chceme použiť funkciu **set_unexpected()**, musíme včleniť hlavičkový súbor **<exception>**. Jednoduché použitie tejto vlastnosti ilustruje nasledovný príklad:

```
// Špecifikácia výnimky & unexpected()
#include <exception>
#include <iostream>
#include <cstdlib>
using namespace std;

class A{};
class B{};
void Test();

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

// void Test() {} // Verzia 1
void Test() { throw 47; } // Verzia 2

void moje_unexpected() {
  cout << "vyslana neocakavana vynimka" << endl;
  exit(0);
}

int main() {
  set_unexpected(moje_unexpected);
  // (ignoruj navratovu hodnotu)
  for(int i = 1; i <=3; i++)
    try {
      Fun(i);
    }
    catch(A) {
      cout << "Zachytena vynimka A" << endl;
    }
    catch(B) {
      cout << " Zachytena vynimka B" << endl;
    }
}
```

Triedy **A** a **B** sú vytvorené iba na vyslanie výnimiek. Triedy výnimiek sú často malé, ale určite môžu uchovávať doplňujúce informácie, na ktoré sa obslužné rutiny môžu dotazovať.

Funkcia **Fun()** tvrdí vo svojej špecifikácií výnimiek, že bude vysielať iba výnimky typu **A** a **B,** a keď sa pozrieme na definíciu funkcie, zdá sa to byť hodnoverné. Prvá verzia funkcie **Test()**, ktorú funkcia **Fun()** volá, nevysiela žiadne výnimky, takže je to pravda. Ale ak niekto zmení funkciu **Test()** tak, že bude vysielať nejaký iný typ výnimky (ako napríklad druhá verzia funkcie, ktorá vysiela výnimku typu **int**), špecifikácia výnimiek funkcie **Fun()** nie je dodržaná.

Funkcia **moja_unexpected()** nemá žiaden argument alebo návratovú hodnotu, čím spĺňa požiadavky na užívateľskú **unexpected** funkciu. Jednoducho iba zobrazuje oznam, aby sme videli, že bola volaná a ukončí program.

Vo funkcii **main()** je **try** blok v rámci slučky **for**, takže sa otestujú všetky možnosti. Takýmto spôsobom môžeme dosiahnuť niečo v zmysle obnovenia stavu. **try** blok vložíme do vnútra cyklu **for**, **while**, **do** alebo **if** a pri akejkoľvek výnimke sa pokúsime problém vyriešiť. Potom opäť skúsime **try** blok.

Program zachytáva iba výnimky typu **A** a **B**, pretože sú to jediné výnimky, o ktorých bol programátor oboznámený, že budú vysielané. Druhá verzia funkcie **Test()** spôsobí zavolanie funkcie **moja_unexpected()**, pretože funkcia **Fun()** vysiela výnimku typu **int** (nepriamo, prostredníctvom volania funkcie **Test()**).

Vo volaní funkcie **set_unexpected()** sa návratová hodnota ignoruje, ale mohli by sme ju tiež uložiť do smerníka na funkciu a neskôr obnoviť ako sme to urobili v príklade použitia **set_terminate()**.

Typická obslužná rutina **unexpected** zvyčajne zaprotokoluje chybu a ukončí program volaním funkcie **exit()**. Avšak okrem toho môže vyslať ďalšiu výnimku (alebo znovu vyslať tú istú výnimku) alebo zavolať funkciu **abort()**. Ak vyšle výnimku typu, ktorý povoľuje funkcia, ktorej špecifikácia bola pôvodne narušená, vyhľadávanie obslužnej rutiny začne v mieste volania funkcie, ktorá vyslala neočakávanú výnimku. (Takéto správanie je jedinečné pre funkciu **unexpected()**).

Ak výnimka vyslaná z obslužnej rutiny **unexpected** nie je pôvodnou špecifikáciou funkcie dovolená, nastane jedna z nasledovných možností:

1. Ak bola v špecifikácií výnimiek funkcie uvedená výnimka typu **std::bad_exception** (definovaná v súbore **<exception>**, výnimka vyslaná z funkcie **unexpected** obslužnej rutiny sa nahradí objektom **std::bad_exception** a vyhľadávanie sa obnoví od funkcie ako predtým.
2. Ak pôvodná špecifikácia výnimiek funkcie neobsahuje výnimku typu **std::bad_exception**, zavolá sa funkcia **terminate()**.

Takéto správanie ilustruje nasledujúci program:

```
// Nesprávna výnimka
#include <exception>    // deklarácia std::bad_exception
#include <iostream>
#include <cstdio>
using namespace std;

// Výnimkové triedy
class A {};
class B {};

// terminate() rutina
void moj_term_handler() {
  cout << "Volane terminate()" << endl;
  exit(0);
}

// obsluha unexpected()
void moj_unexpected1() {
  throw A();
}
void moj_unexpected2() {
  throw;
}

// Ak by sme da;i tento throw prikaz do Fun alebo FunX,
```

```
// kompilator by detekoval chybu, takze to dame do vlastnej funkcie
void FunVynimka() {
  throw B();
}

void Fun() throw(A) {
  FunVynimka();
}
void FunX() throw(A, bad_exception) {
  FunVynimka();
}

int main() {
  set_terminate(moj_term_handler);
  set_unexpected(moj_unexpected1);
  try {
    Fun();
  }
  catch (A&) {
    cout << "Zachytene A z funkcie Fun" << endl;
  }
  set_unexpected(moj_unexpected2);
  try {
    FunX();
  }
  catch (bad_exception&) {
    cout << "Zachytene bad_exception z FunX" << endl;
  }
  try {
    Fun();
  }
  catch (...) {
    cout << "Toto sa nikdy nevytlaci" << endl;
  }
}
```

Obslužná rutina **moj_unexpected1** vysiela akceptovateľnú výnimku (**A**), takže vykonávanie sa pri prvom zachytení obnoví. Obslužná rutina **moj_unexpected2** nevysiela správnu výnimku (**B**), ale pretože funkcia **FunX()** uvádza vo svojej špecifikácií výnimiek výnimku **bad_exception**, výnimka **B** sa nahradí objektom **bad_exception** a druhé zachytenie bude tiež úspešné. Pretože funkcia **Fun()** neobsahuje vo svojej špecifikácií výnimku **bad_exception**, zavolá sa ako ukončovacia rutina funkcia **moj_term_handler()**.

Kedy *nepoužívať* špecifikáciu výnimiek

Ak sa pozrieme na deklarácie funkcií v štandardnej C++ knižnici, nenájdeme ani jednu špecifikáciu výnimky. Hoci sa toto zdá divné, je na to dobrý dôvod: knižnica pozostáva hlavne zo šablón, a nikdy nevieme, čo generický typ urobí. Predpokladajme napríklad, že vytvárame generickú zásobníkovú šablónu a k funkcii **pop()** sa pokúsime pripojiť špecifikáciu výnimky, napríklad takto:

```
T pop() throw(logic_error);
```

Pretože jedinou chybou, ktorú očakávame je podtečenie zásobníka, mohli by sme si myslieť, že je bezpečné špecifikovať **logic_error** alebo nejaký iný vhodný typ výnimky. Ale pretože nevieme nič o type **T**, čo ak kopírovací konštruktor bude vysielať nejakú výnimku (nie je to nakoniec nič nerozumné). Potom sa zavolá funkcia **unexpected()** a náš program skončí. Ak nevieme aké výnimky môžu nastať, špecifikácie výnimiek nepoužívajme. A preto šablónové triedy, ktoré predstavujú 90 % štandardnej C++ knižnice nepoužívajú špecifikácie výnimiek - použité výnimky sa uvádzajú v dokumentácií a zvyšok je ponechaný na užívateľa. Špecifikácie výnimiek sú určené hlavne pre ne-šablónové triedy.

Programovanie s výnimkami

Väčšina programátorov, najmä C programátorov, výnimky zo svojho jazyka nepozná. A preto niekoľko pravidiel pre programovanie s výnimkami.

Výnimku nie sú odpoveďou na všetky problémy. Nasledujúce časti zdôrazňujú situácie, kde výnimky nie sú zaručené. Pravdepodobne najlepšou radou pri rozhodovaní, kedy vyslať výnimku je, že len vtedy, keď funkcia zlyhá plniť svoju špecifikáciu.

Nie pre asynchrónne udalosti

C++ výnimky nemôžeme použiť na ošetrenie asynchrónnych udalostí, pretože výnimka a jej ošetrenie nie sú v tom istom zásobníku volaní. Znamená to, že výnimky sa spoliehajú na dynamickú reťaz volaní funkcií programového zásobníka (dynamický rozsah), zatiaľ čo asynchrónne udalosti sa musia spracovávať v celkom oddelenom kóde, ktorý nie je súčasťou normálneho toku programu (typicky, obslužné rutiny prerušení alebo cyklus udalostí).

Nechceme však tým povedať, že asynchrónne udalosti nemôžu byť združené s výnimkami, ale obsluha prerušenia by mala robiť svoju prácu čo najrýchlejšie a hneď vrátiť riadenie. Neskôr na vhodnom mieste programu sa môže vyslať výnimka, vychádzajúca z prerušenia.

Nie pre mierne chybové podmienky

Ak je dostatok informácií na ošetrenie chyby, nie je to výnimka. Radšej sa postarajme o chybu v aktuálnom kontexte, než aby sme vysielali výnimku do rozsiahlejšieho kontextu.
C++ výnimky sa tiež nevysielajú pre udalosti na strojovej úrovni, napríklad delenie nulou. Predpokladá sa, že s týmito udalosťami sa vysporiada nejaký iný mechanizmus, napríklad operačný systém alebo hardvér. Takýmto spôsobom C++ výnimky môžu byť náležíte efektívne a ich použitie je izolované iba na výnimkové stavy na úrovni programu.

Nie pre riadenie priebehu programu

Výnimka vypadá ako alternatívny návratový mechanizmus a čiastočne ako **switch** príkaz, takže nás to láka používať výnimku inak, než je jej pôvodný zámer. Toto je zlý nápad, čiastočne pretože obslužný systém výnimiek je omnoho menej výkonný, než normálne vykonávanie programu. Výnimky sú zriedkavé udalosti, takže normálny program by nemal za ne platiť. Zároveň výnimky použité inak, než pre chybové stavy, sú pre užívateľa triedy alebo funkcie značne mätúce.

Výnimky nie sme nútení používať

Niektoré programy sú celkom jednoduché (napríklad drobné utility). Potrebujeme zadať iba nejaký vstup a vykonať nejaké spracovanie. V takýchto programoch by sme sa mohli pokúšať alokovať pamäť a zlyhať, pokúšať sa otvoriť súbor a zlyhať, atď. V takýchto programoch je prípustné použiť funkciu **assert()** (alebo nejakú ekvivalentnú funkciu) alebo zobraziť oznam a program ukončiť, a dovoliť tak systému urobiť poriadok namiesto namáhavého zachytávania výnimiek a obnovovania prostriedkov. Ak nepotrebujeme použiť výnimky, nepoužívajme ich.

Nové výnimky, starý kód

Ďalšou situáciou, ktorá môže nastať, je modifikácia existujúceho programu, ktorý nepoužíva výnimky. Mohli by sme napríklad pridávať knižnicu, ktorá ***používa*** výnimky a byť zvedaví, či potrebujeme modifikovať celý náš kód programu. Za predpokladu, že máme akceptovateľnú schému ošetrovania chýb, najmúdrejšie je obaliť najväčší blok, ktorý používa novú knižnicu (týmto môže byť celý kód funkcie **main()**) **try** blokom, za ktorým bude nasledovať **catch(...)** a základný chybový oznam. Toto môžeme do ľubovoľnej potrebnej úrovne vylepšiť pridaním presnejších obslužných rutín, ale v každom prípade, kód, ktorý budeme nútení pridať bude minimálny.

Tiež by sme mohli izolovať náš výnimky generujúci kód v **try** bloku a napísať obslužné rutiny, čím výnimky skonvertujeme do vlastnej existujúcej schémy ošetrovania chýb.

Premýšľať o výnimkách je skutočne dôležité, najmä ak vytvárame knižnicu pre niekoho iného, kto ju bude používať, obzvlášť v situáciách, v ktorých nemôžeme vedieť aká má byť odozva na kritické chybové stavy.

Typické použitie výnimiek

Výnimky by sa mali používať v nasledujúcich situáciách:

- Vyriešenie problému a opätovné zavolanie funkcie ktorá výnimku spôsobila.
- Zorganizovanie vecí a pokračovanie bez opätovného volania funkcie.
- Výpočet nejakého alternatívneho výsledku namiesto toho, čo mala poskytovať funkcia.
- Urobenie hocičoho v aktuálnom kontexte a znovu vyslanie ***tej istej*** výnimky do vyššieho kontextu.
- Urobenie hocičoho v aktuálnom kontexte a znovu vyslanie ***odlišnej*** výnimky do vyššieho kontextu.
- Ukončenie programu
- Obaľovacie funkcie (najmä C knižničné funkcie, ktoré používajú obyčajnú chybovú schému tak, aby poskytovali výnimky).
- Zjednodušenie. Ak schéma výnimiek robí veci komplikovanejšími, je bolestivé a nepríjemné používať ju.
- Vytvorenie bezpečnejšej knižnice a programu. Toto je dlhodobá investícia (odolnosť aplikácie).

Kedy používať špecifikácie výnimiek

Špecifikácia výnimky je ako prototyp funkcie: oznamuje užívateľovi, aby napísal kód na ošetrenie výnimiek a aké výnimky má ošetrovať. Kompilátoru oznamuje, aké výnimky môžu z tejto funkcie vychádzať, takže dokáže detekovať porušenia počas chodu programu.

Samozrejme, nemôžeme sa neustále pozerať na kód a zisťovať, aké výnimky sa v konkrétnej funkcii objavia. Občas funkcie, ktoré zavoláme, vysielajú neočakávanú výnimku a občas stará funkcia, ktorá nevysielala žiadnu výnimku je nahradená novou, ktorá nejakú výnimku vysiela, a zavolá sa funkcia **unexpected()**. Vždy keď použijeme špecifikácie výnimiek alebo voláme funkcie, ktoré ju obsahujú, uvažujme o vytvorení vlastnej funkcie **unexpected()**, ktorá bude správu zaprotokolovať a znovu vyšle tú istú výnimku.

Ako sme už povedali, špecifikáciám výnimiek by sme sa mali vyhýbať v šablónových triedach, pretože nedokážeme zistiť, aké typy výnimiek by mohli vysielať parametrické triedy šablóny.

Začnime so štandardnými výnimkami

Skôr ako začneme používať vlastné výnimky, pozrime sa na štandardné C++ výnimky. Ak štandardná výnimka spĺňa naše požiadavky, je pravdepodobne omnoho jednoduchšie pre užívateľa pochopiť ju a použiť.

Ak výnimka, ktorú chceme použiť, nie je súčasťou štandardnej knižnice, snažme sa odvodiť vlastnú výnimku zo štandardnej výnimky. Je pekné, keď užívatelia môžu písať kód tak, že majú k dispozícií funkciu **what()**, definovanú v rozhraní triedy **exception**.

Vnárajme svoje vlastné výnimky

Ak vytvárame výnimky pre konkrétnu triedu, je dobré vnárať triedy výnimiek buď do vnútra triedy alebo do vnútra menového priestoru, obsahujúceho triedu, aby užívateľovi bolo jasné, že táto výnimka je použitá len v rámci triedy. Okrem iného takto predídeme i znečisteniu globálneho menového priestoru.

Vlastné výnimky môžeme vnárať i v prípade, že sú odvodené zo štandardných C++ výnimiek.

Používajme hierarchie výnimiek

Používanie hierarchií výnimiek je významným spôsobom klasifikácie typov kritických chýb, na ktoré môže trieda alebo knižnica naraziť. Toto poskytuje užívateľovi prospešné informácie, napomáha pri organizovaní jeho kódu a poskytuje mu možnosť ignorovať všetky špecifické typy výnimiek a zachytávať iba výnimku typu základnej triedy. Zároveň pridanie akejkoľvek výnimky, zdedenej z tej istej základnej triedy nás neprinúti prepisovať všetok existujúci kód - obslužná rutina základnej triedy zachytí i novú výnimku.

Dobrým príkladom hierarchie výnimiek sú štandardné C++ výnimky.

Zachytávajme odkazom, nie hodnotou

Už sme vysvetľovali pri hľadaní zhody výnimiek, výnimky by sa mali zachytávať odkazom z dvoch dôvodov:

- Aby sme sa vyhli vytváraniu zbytočnej kópie objektu výnimky pri jeho prenose do obslužnej rutiny
- Aby sme sa vyhli orezaniu objektu pri zachytávaní odvodenej výnimky ako objektu základnej triedy

Hoci by sme tiež mohli vysielať a zachytávať smerníky, týmto by sme zaviedli väčšiu previazanosť kódu - vysielajúci a zachytávajúci sa musia dohodnúť ako bude objekt výnimky alokovaný a následne upratený. Toto je problém, pretože samotná výnimka môže byť spôsobená vyčerpaním haldy. Ak vysielame objekty výnimky, o pamäťové záležitosti sa stará systém ošetrenia výnimky.

Vysielajme výnimky v konštruktoroch

Pretože konštruktor nevracia žiadnu hodnotu, k dispozícií sme mali doteraz dve možnosti ako oznámiť chybu počas konštrukcie:

- Nastaviť nelokálny indikátor a dúfať, že užívateľ ho skontroluje
- Vrátiť neúplne vytvorený objekt a dúfať, že užívateľ ho skontroluje

Toto je opäť závažný problém, pretože C programátori sa spoliehali na implicitnú záruku, že tvorba objektu bude vždy úspešná, čo nie je v C nerozumné, pretože typy sú tu veľmi jednoduché. Avšak pokračovanie vo vykonávaní C++ programu po zlyhaní konštrukcie je zárukou katastrofy, takže konštruktory sú jedným z najdôležitejších miest, kde sa majú výnimky vysielať. Výnimky poskytujú bezpečný a efektívny spôsob ošetrenia chýb konštruktora. Avšak keď sa vysielajú výnimky vo vnútri konštruktora, potom súčasne musíme tiež dávať pozor na smerníky vo vnútri objektov a spôsob ich upratovania.

Negenerujme výnimky v deštruktoroch

Pretože deštruktory sa volajú v procese vysielania iných výnimiek, nikdy nesmieme vyslať výnimku v deštruktore alebo zapríčiniť vyslanie inej výnimky nejakou činnosťou, ktorú v deštruktore vykonávame. Ak to nastane, môže byť nová výnimka vyslaná *pred* dosiahnutím **catch** klauzuly existujúcej výnimky, čo spôsobí zavolanie funkcie **terminate()**.

Ak zavoláme akúkoľvek funkciu vo vnútri deštruktora, ktorá môže vysielať výnimky, takéto volania by mali byť v deštruktore v rámci **try** bloku a deštruktor musí ošetriť všetky výnimky sám. Z deštruktora nesmie uniknúť žiadna výnimka.

Réžia použitia výnimiek

Vyslanie výnimky predstavuje značnú réžiu chodu programu (ale je to ***dobrá réžia***, pretože objekty sa upracú automaticky). Z tohto dôvodu by sme výnimky nemali používať ako súčasť normálneho toku programu, bez ohľadu na to ako veľmi nás to pokúša a aké šikovné sa to zdá. Výnimky by sa mali vyskytovať iba zriedka, takže réžia sa nahromadí iba výnimočne a nie pri normálnom vykonávaní kódu. Jedným z dôležitých cieľov pri návrhu ošetrovania výnimiek bolo i to, aby sa dali implementovať bez vplyvu na rýchlosť vykonávania programu, ak nebudú použité, t.j. ak nevyšleme výnimku, kód bude pracovať tak rýchlo ako by pracoval bez ošetrovania výnimiek. Či tomu je tak závisí od implementácie konkrétneho prekladača.

Výraz **throw** si môžeme predstaviť ako volanie špeciálnej systémovej funkcie, ktorej argumentom je objekt výnimky, a ktorá sa vracia hore cez reťaz vykonávaní. Aby sa toto mohlo robiť, kompilátor musí ukladať do zásobníka extra informácie, ktoré pomáhajú pri odvíjaní zásobníka. Aby sme to pochopili, potrebujeme niečo vedieť o programovom zásobníku. Zakaždým, keď sa zavolá funkcia, informácia o tejto funkcii sa ukladá do programového zásobníka ako inštancia aktivačného záznamu (*activation record instance* - ARI), tiež nazývaný *zásobníkový rámec* (stack frame). Typický zásobníkový rámec obsahuje adresu volajúcej funkcie (aby sa vykonávanie mohlo vrátiť), smerník na ARI statického rodiča funkcie (rozsah, ktorý lexikálne

obsahuje volanú funkciu, aby bol prístup ku globálnym premenným) a smerník na funkciu, ktorá ju volala (*dynamický rodič*). Cesta, ktorej logickým výsledkom opakovaného sledovania liniek dynamických rodičov je *dynamická reťaz* alebo *reťaz volaní*. Takýmto spôsobom môže vykonávanie postupovať späť pri vyslaní výnimky, a je to mechanizmus, ktorý umožňuje komponentom, vytvoreným bez poznania jeden druhého, aby si navzájom posielali chyby za chodu programu.

Aby bolo umožnené odvíjanie zásobníka pri ošetrovaní výnimky, každý zásobníkový rámec musí obsahovať o každej funkcii extra informácie, týkajúce sa výnimky. Táto informácia popisuje, ktoré deštruktory sa musia zavolať (aby sa upratali lokálne objekty), indikuje, či aktuálna funkcia obsahuje **try** blok a vymenúva aké výnimky priradené **catch** bloky dokážu spracovávať. Prirodzene za túto informáciu naviac platíme priestorom, takže programy, ktoré podporujú spracovanie výnimiek sú o čosi väčšie, než tie, ktoré výnimky nepodporujú[3]. Dokonca i veľkosť programov počas kompilácie, používajúcich ošetrovanie výnimiek, je väčšia, pretože kompilátor musí generovať i logiku, ktorá generuje rozšírený zásobníkový rámec za chodu programu.

napadne nás, že toto extra hospodárenie spomaľuje vykonávanie, a je to pravda. Avšak chytré implementácie kompilátorov sa dokážu tejto réžií vyhnúť. Pretože informácie o kóde ošetrovania výnimiek a ofsety lokálnych objektov sa dajú vypočítať naraz počas kompilácie, takáto informácia sa dá uchovávať na jednom mieste, priradenom ku každej funkcii a nie v každom ARI. V podstate odstránime réžiu výnimiek z každého ARI, a preto sa zbavíme extra času, potrebného na ukladanie do zásobníka. tento prístup sa nazýva model s *nulovou réžiou ošetrovania výnimiek*[4] a optimalizované ukladanie do pamäti sa nazýva *tieňový zásobník*..

Zhrnutie

Zotavenie po chybe je základným záujmom každého programu, a obzvlášť dôležité je v jazyku C++, ktorého jedným z cieľov je vytvárať programové komponenty pre ďalšie použitie. Ak chceme vytvoriť odolný systém, odolný musí byť každý komponent.

Cieľom ošetrovania výnimiek v C++ je zjednodušenie tvorby rozsiahlych spoľahlivých programov použitím menej kódu, než je momentálne možné s väčšou istotou, že aplikácia nebude obsahovať neošetrenú chybu. Toto sa dosahuje za cenu o čosi alebo vôbec zníženej výkonnosti a malého vplyvu na existujúci kód. Základné výnimky sa ľahko učia a v programe môžeme s nimi začať kedykoľvek. Výnimky sú jednou z tých vlastností, ktoré poskytujú okamžitý a značný úžitok pre projekt.

[3] Toto samozrejme závisí od toho, koľko kontrol návratových kódov by sme museli vložiť do kódu, ktorý nepoužíva výnimky.
[4] Kompilátor GNU C++ používa implicitne model s nulovou réžiou.