A programozás alapjai 2.

Hivatalos laborsegédlet hallgatók számára.

Miért használjunk kivételkezelést?

A program futása során keletkező problémák, hibák kezelésére egyszerű és tiszta lehetőségeket biztosít, és használatukkal kevésbé valószínű, hogy egy hibát nem kezelünk le. Hiba esetén a vezérlés a hibakezelő részre ugrik.

A régi módszer a hibakódok visszaadása és hibaüzenetek kiírása több szempontból is előnytelen. Ekkor keverednek a programunk funkcionalitását megvalósító kódrészek a hibakezelő kódrészekkel. Ennek következtében a kód zavaros lesz, nehéz lesz karbantartani és nagy a valószínűsége az elfelejtett hibakezelésnek. Konstruktorban például más módon lehetetlen is hibát jelezni, hiszen nem tud hibakódot visszaadni.

A hibakódok kezelése nem jól skálázódik, az exception-használat viszont igen. Ez egy 10 soros kódban nem fog látszódni, de 10000 sornál már látszódik a különbség.

Gyakori, hogy a hibát nem abban a függvényben kell kezelni, ahol ténylegesen keletkezett, hanem egy hosszú hívás lánc másik végén. Ezt exceptionökkel könnyű kezelni, viszont a hibakódok vizsgálatát minden egyes sziten el kell végezni, ami ráadásul teljesen felesleges. A különbséget az alsó táblázatban látható példakódok szemléltetik:

Exception	Error code

`void f1() {`
`  try {`
`  // ...`
`  f2();`
`  // ...`
`  } catch (const some_exception& e) {`
`  // ...hiba kezelése...`
`  }`
`}`
`void f2() { ...; f3(); ...; }`
`void f3() { ...; f4(); ...; }`
`void f4() { ...; f5(); ...; }`
`void f5() {`
`  // ...`
`  if (/*...hiba feltétel...*/)`
`  throw some_exception();`
`  // ...`
`}`
	`int f1() {`
	`  // ...`
	`  int rc = f2();`
	`  if (rc == 0) {`
	`  // ...`
	`  } else {`
	`  // ...hiba kezelése...`
	`  }`
	`}`
	`int f2() {`
	`  // ...`
	`  int rc = f3();`
	`  if (rc != 0)`
	`  return rc;`
	`  // ...`
	`  return 0;`
	`}`
	`int f3() {`
	`  // ...`
	`  int rc = f4();`
	`  if (rc != 0)`
	`  return rc;`
	`  // ...`
	`  return 0;`
	`}`

	`int f4() {`
	`  // ...`
	`  int rc = f5();`
	`  if (rc != 0)`
	`  return rc;`
	`  // ...`
	`  return 0;`
	`}`
	`int f5() {`
	`  // ...`
	`  if (/*...hiba feltétel...*/)`
	`  return some_nonzero_error_code;`
	`  // ...`
	`  return 0;`
	`}`

Viszont soha ne használjunk exceptiont visszatérési érték átadására!

Az exception használata

A **throw** kulcsszóval tudunk exceptiont dobni bárhol a kódban, amikor egy probléma felmerül. A **catch**-csel jelölt blokkban kaphatjuk el és kezelhetjük a dobott exceptiont. A **try** egy védett blokkot jelent. Az itt keletkező hibákat a try blokkot követő catch blokk(ok)ban kezeljük.

Ha a hívási fában találunk egy hibafeltételt, throw-val hibát dobunk. A throw-nak paramétert kell adni, ami lehet beépített típusú változó vagy tetszőleges osztálybeli objektum. A throw olyan, mint egy return, addig ugrál felfelé a hívási fában, amíg egy megfelelő catch el nem kapja (aminek a paramétere kompatibilis a dobott típussal).

A catch blokkban definiált kódrész csak akkor fut le, ha történt hiba. Ha azt szeretnénk, hogy minden hibát elkapjon, akkor ezt a catch(...) szintaxissal tehetjük meg. Ennek a használata viszont a hibakezelési elveket betartva ritka, mivel információt kell szolgáltatnunk a felhasználónak a hiba keletkezésének okáról és lehetséges megoldásának módjáról. Ezt viszont nehéz úgy megtenni, hogy nem tudjuk, milyen hibát kaptunk el.

Ha nem keletkezik hiba a try blokkban, vagy hiba esetén lefut egy catch blokk, a végrehajtás a try-catch blokk utántól folytatódik. Try-catch blokkot helyezzünk minden olyan kódrész köré, ami hibát dobhat, és az adott helyen azt szeretnénk kezelni.

Példa

```
try {
   // védett kód
   func1();
   func2();
} catch( const ExceptionName1& e1 ) {
   // catch block 1 ...
} catch( const ExceptionName2& e2 ) {
   // catch block 2 ...
} catch( const ExceptionName3& eN ) {
   // catch block N ...
}
func3();
```

A fenti kódrészletben ha a try blokk függvényhívásai során nem keletkezik hiba, a catch blokkok lefutása nélkül a program végrehajtása a func3() hívásra ugrik. Tehát lefut a func1(), func2(), func3().

Tegyük fel, hogy most a func1() függvényben exception dobódik. Ekkor a func1() azonnal visszatér, és a func2() lefutása nélkül a megfelelő catch blokkra ugrik. Ha viszont egy ExceptionName2 típusú hiba keletkezett, a második catch blokk fog lefutni, ahol pl. kiírjuk a megfelelő hibaüzenetet a felhasználónak. Ezt követően a végrehajtás a func3() hívással folytatódik. Tehát lefut a func1() valamennyi része (a hibáig), a 2. catch blokk és a func3().

Egymásba ágyazás és hiba újradobása

Lehetőségünk van try-catch blokkok egymásba ágyazására, tehát egy try blokkon belül lehet másik try blokk is. A belső catch blokkból tovább tudunk dobni egy hibát a throw; szintaxissal. Ekkor nem kell paramétert megadni a throw-nak. Ilyet csak catch blokkon belül tehetünk meg. Catch blokkon belül is lehet akár új hibát is dobni, de ha egy try blokkhoz tartozó valamelyik catch blokkba már beléptünk, akkor az ugyanahhoz a try blokkhoz tartozó másik catch blokkba már nem fog belépni.

Ennek legfőképp akkor van gyakorlati haszna, ha a belső try-catch blokkban bekövetkező hiba esetén is szeretnénk lefuttatni egy kódrészletet, de azt is védett try-catch blokkban.

Például:

```
try {
  try {
      // kód
  }
  catch (int n) {
      throw;
  }
  // kód
}
catch (int e) {
  // hiba kezelés
}
```

Típusosság

A C++ kivételek típusosak, a típusuk alapján tudjuk elkapni őket. Hiba esetén a try blokk utáni catch blokkok sorrendben lesznek ellenőrizve, hogy a paraméter típusa kompatibilis-e a hiba típusával.

Megáll a végrehajtás és belép a catch blokkba, ha catch(...)-ot talál, ami mindent megfog. Nagy hátránya, hogy nem kapjuk meg paraméterben az exception objektumot.

A catch paraméter típusa egyezhet pontosan a kivétel objektum típusával (int, double, Person). Ez úgy viselkedik, mint egy paraméterátadás függvénynek, másolat készül az objektumról.

A catch paramétere lehet referencia vagy konstans referencia ugyanarra a típusra vagy az osztály ősére. Ezt a megoldást preferáljuk általában, mert például 20 catch blokkot le tudunk fedni egy catch blokkal, ami egy olyan paraméterű hibaüzenetet vár, amiből az előző 20 catch blokk paramétere öröklődik.

Ennek megfelelően van egy beépített osztályunk (std::exception), amiből sok más, hasznos exception osztály öröklődik. Ezeket itt találjátok, érdemes jó alaposan megnézni a listát:
http://www.cplusplus.com/reference/exception/exception/

Vegyük például az std::out_of_range hibaosztályt. Akkor szokás használni, ha valamilyen adatokat tároló struktúrát (például vektort) túl, illetve alul indexelünk. Laboron többször volt ilyen szituáció, amit így kellett volna megoldani.

Fontos, hogy ha saját hibaosztályt szeretnénk létrehozni, akkor érdemes az std::exceptionből örököltetni, hiszen ha valaki olyan try-catchet ír, ami std::exceptiont kap el, jó okkal feltételezhetjük, hogy a mi custom exceptionünket is el szeretné kapni. Egyébként nem érdemes konkrétan std::exceptiont elkapni, mert az olyan, mintha a szőnyeg alá söpörnénk a problémát.

Triviális, de a félreértés elkerülése végett gondoljuk ezt át: mivel a hibaosztályok is osztályok, nem csak egy szintes, de akár többszintes öröklési fát is készíthetünk velük. Ekkor a fa teteje az std::exception, majd ahogy haladunk le a fában, egyre hibaspecifikusabb hibaosztályokkal találkozhatunk., például: std::exception → network_error → packet_error → invalid_padding_format.

Általános gyakorlat, hogy adat osztályokat (például Person, Vehicle stb.) nem dobunk és nem kapunk el, mert nem hibát képviselnek. Emellett számokkal se szokás dobálózni (pl. throw 5;), mert éppen az volt a célunk az egésszel, hogy megszabaduljunk a semmitmondó hibakódoktól. Ezek helyett érdemes például egy mappát készíteni a projektünkön belül és oda megírni a különböző hibákat reprezentáló hiba osztályainkat (network_error, packet_error stb.), amik persze tartalmazhatnak extra változókat, mint például egy hibaüzenetet vagy érintett változónevet stb. Ezáltal a kód is sokkal olvashatóbbá válik és mindig tudni fogjuk, hogy mit tudunk egyáltalán elkapni és mit nem (nem kell felkészülnünk a 40 féle adat osztályunk repkedésére).

Alapvető szabály, hogy amit dobunk, arról mindig másolat készül, még ha referenciával kapjuk is el a kivétel objektumot. Ennek előnye, hogy nem egy lokális objektumra kapunk referenciát. Viszont ha pointert dobunk egy lokális változóra, akkor az egy rossz megoldás. Ha esetleg new-val hoznánk létre a kivétel objektumot, akkor abba a problémába futhatunk, hogy mikor és ki fogja felszabadítani. Elkapáskor használjunk referenciát vagy konstans referenciát.

A stack kezelése

Amennyiben egy hosszabb hívási láncban keletkezik kivétel, akkor megkezdődik a kivételt elkapó catch keresése. A hívási láncban visszafele lépkedve, ha nem találunk az adott függvényben catch blokkot, ami elkapná a hibát, még egy szintet visszalépünk. Visszalépés előtt viszont a függvény minden lokális változója felszabadul. Természetesen mindnek meghívódik a konstruktora is. Figyelem, a dinamikusan foglalt memóriaterületek nem lesznek felszabadítva!

Destruktorban soha ne dobjunk olyan kivételt, amit nem kapunk el rögtön a destruktorban. Ha egy kivétel közben a destruktorban újabb kivételt dobnánk (amit nem a destruktorban kapunk el), akkor az alkalmazás kilép.