A Programozás Alapjai 2
Objektumorientált szoftverfejlesztés

Dr. Forstner Bertalan
forstner.bertalan@aut.bme.hu

BME AUT
Department of Automation and Applied Informatics

Konverziók

Típuskonverzió

- Ha a függvény/operátor paraméter típusa eltérő,
- Ha a függvény/operátor visszatérési érték típusa eltérő
- Ha az inicializálandó és az inicializáló objektum típusa eltérő
- Egyéb, összetett kifejezésekben

- típuskonverzióra lehet szükség

Példa

```
double f(double d) {
        int x = 2;
        d = x; // automtaikus konverzió operátornál
        return x; // itt egy aut. konv a visszatérési érték
}

void main() {
        f(10); // itt egy aut. konv. a paraméter
}
```

- Létrejön egy temporális objektum amit aztán pl. fent az operator= megkap.

Automatikus konverzió

- Megnézzük:
  - Beépített típusokra
  - Szülő pointerrel leszármazottra
  - Leszármazottról szülőre
  - Két, nem kapcsolódó típusra

Példa beépített típusokra

- Mind a három konverzió ugyanazt csinálja:

```
int x = 2;


double d1 = x;  // működik az aut. konverzió itt
double d2 = (double) x; // ki is írhatjuk
double d3 = double(x);  // ki is írhatjuk, „konstruktor hívás”
```

Beépített típusokra

- Általában kisebb méretűből nagyobbra van automatikus konverzió, vagyis ahol nem veszíthetünk adatot.
- Fordítva:

```
double d4 = 3.13;
int y1 = d4; // warning: possible loss of data
int y2 = (int)d4;
int y3;
y3 = d4; // warning: possible loss of data
```

Ha nem szeretnénk warningot látni, castoljunk explicit.

Referenciára

- Referenciára nincs automatikus cast, még akkor sem, ha bővebb típusra is castolunk.

```
void f(double& d) {d=1;}

void g() {
        int m=2;
        f(m);            // hiba
        f((double)m);    // hiba
        f((double&)m);   // lefordul, de …
}
```

Referenciára

```
f((double&)m);    // lefordul, de …
```

- Öngyilkosság: **nem születik** új változó (bár ha születne egy temporális, arra referenciát atadni is őrültség lenne),
- Az **eredeti** intre lesz egy double referencia: **túlcímzi** a területet, amikor állítja a double változót.
- Olyan, **mintha int*-ot** adnánk át és double*-ként értelmezve használnánk.

Referenciára

- Ellenben ha konstans referenciát veszünk át:

```
void f(const double& d) {d=1;}
```

- Úgy OK:
  - létrehoz egy temporális változót az új típusból, és arra referenciát ad át.

Szülőpointerrel/referenciával leszármazottra

- Van automatikus konverzió
- Láttuk a virtuális tagfüggvényeknél
- Ha A szülője B-nek:

```
B b;
A* pa = &b;
```

Leszármazottból szülőre, de nem pointerrel

```
A a; //ős

B b; //leszármazott

a = b; // OK,

a = (A)b; // ki is írható,
          //ezt jelenti az előző
```

Slicing on the fly

Leszármazottból szülőre, de nem pointerrel

```
A a; //ős

B b; //leszármazott

a = b; // OK,

a = (A)b; // ki is írható,
          //ezt jelenti az előző
```

Szülőről a gyerekre nem működik: mivel is egészítené ki?

Két nem kapcsolódó típusra

```
Stack s;

Person p;

s = (Stack)p;
```

- Nem fordul le
  - Mit is csinálna a fordító (Személy osztály objektumából Stack-et)

Két nem kapcsolódó típusra pointerrel

- Pointerrel lehet, de veszélyes, és nem biztos, hogy van értelme:

```
Person jozsi;
Person* pjozsi = &jozsi;
Stack* ps = (Stack*) pjozsi;
// OK, meggyőztük
ps->push();
```

- Egy személy objektumnak mondjuk, hogy push
- => elszáll

Nyomjad, Józsi!

Két nem kapcsolódó típusra

- Van, amikor van értelme automatikus konverziónak nem a hierarchiában is
  - pl. char* és egy String osztály között
  - Vagy egy Complex szám és egy double között

```
void f1(string s) {…}
char* pc = "aaaa";
f1( pc );

void f2(char* s) {…}
String s1("abc");
f2( s1 );

void f3(Complex c) {…}
double d = 87;
f3( d );
```

Két nem kapcsolódó típusra

- Hogy lehet ezt elérni?
- 2 módszer létezik:
  - konverziós konstruktor
  - konverziós operátor

1. Konverziós konstruktor

- Minden egyargumentumú konstruktor.
  - vagy lehet többargumentumú is, csak a többinek legyen default értéke

```
class Complex {
      double re, im;
public:
      Complex(double d) {re = d; im = 0;}
};
…
Complex c = 5.0;
```

Konverziós konstruktor szabályok

- Csak egy lépés automatikus. Pl. létezik konverzió ilyen módon A-ból B-re, B-ről C-re. Az A→C konverzió nem működik (explicit sem). N-1 lépést írjunk ki explicit.
- Az egy lépésbe a const nem számít bele: ha a konstruktor const-ot vár nem const-tal is lehet hívni.

Konverziós konstruktor szabályok

- Csak egy lépés automatikus. Pl. létezik konverzió ilyen módon A-ból B-re, B-ről C-re. Az A→C konverzió nem működik (explicit sem). N-1 lépést írjunk ki explicit.
- Az egy lépésbe a const nem számít bele: ha a konstruktor const-ot vár nem const-tal is lehet hívni.
- Lehet mégis több, mint egy, de csak egy lépés legyen nem beépített típusra.

Példa

```
class Complex {
      double re, im;
public:
      Complex(double d) {re = d; im = 0;}
};
…
void f(Complex c) {…}
f(10); //működik, ugyanaz, mint:
f((Complex)(double)10);
```

Explicit

- Ha egy konstruktort valóban csak objektum konstruálására szeretnénk alkalmazni, és nem szeretnénk, hogy automatikus konverzióra legyen használva
  - pl. rejtett hibák miatt

Explicit példa

```
class X {
public:
    explicit X(int);
    explicit X(double) {// ...
    }
};
explicit X::X(int) {..}
```

- Tfh van egy függvény, ami X-et vár:

```
void f(X) {}

void g(int i) {
  f(i);              // Hibát fog okozni!
  X x1(i);        // OK – ez sima konstruálás
}
```

2. Konverziós operátor

- Amit a konverziós konstruktor megoldott: egy másik típusból sajátot készített
  - Pl. *char** → *string*,
  - így ha valami *string-et* vár, *char**-al is tudtam hívni.

- Amit a konverziós operátor megold: fordítottja, vagyis egy saját típusból egy más típust készít
  - Pl. *string* → *char**,
  - vagyis ami *char**-ot vár *string-et* is használhasson.
- A konverziós operátor hasonlóan készíthető, mint egyéb operátorok.

Konverziós operátor példa

- A szintaktika: nincs visszatérés és paraméter sem (void sem lehet):

```
class String {
        char* data;
public:
        String( const char* c ) {//…itt a tobbi kihagyva}
        operator char*() const { return data; }
        operator const char*() const { return data; }
};
```

- Használat:

```
String s1("abc");
char* ps1 = s1; // operator char*()-ot hív
const char* ps2 = s1; // operator const char*()
```

Minek a const-os változat?

```
String s1("abc");
char* ps1 = s1; // operator char*()-ot hív
const char* ps2 = s1; // operator const char*()

ps1[5] = 'v';
// Sajnos OK, túlcímezhetek, rendkívül veszélyes

ps2[5] = 'v';
// Szerencsére le sem fordul, mert a ps2 const char*,
// így ps2-n keresztül csak olvasni lehet
```

- Vagyis
- Ha a cél az, hogy csak olvasni lehessen, a megoldás: csak a const verziót írom meg.

Minek a const-os változat?

- Vagyis csak a constot megírva pl. jól működik a következő, strcpy-s példa:

```
String s1("aaa");
char buff[1000] ;
strcpy(buff, s1); // olvasás, on the fly konverzió
```

- Viszont kiszűri a következőt:

```
String s1("aaa");
char* buff="safdsdfsaffaaf" ;
strcpy(s1, buff); // írás túlcímzéssel,
// de szerencsére le sem fordul, mivel s1 const.
```

Konverziós operátor szabályok

- Csak egy lépés automatikus

```
class A { //Egy osztály, ami int-té tud válni
   public:
       operator int() { …}
};

class B { //Egy osztály, ami A-vá tud válni
   public:
       operator A() { … }
};

void f() {
       B b;
       int x = b; // nem jó, nem tud B intté válni
       int x = (A)b; // jó, mivel előbb A lett
```

Konverziós operátor szabályok

- **Öröklődik**
  - Írunk az ősben egyet, ami int-re konvertál. A leszármazottban nem írunk.
  - Ekkor működik a leszármazottra is.
- **Lehet virtuális**
  - Pl. char *-gá konverziót megírunk az ősben virtuális tagfüggvényuként, és a leszármazottban felüldefiniáljuk.
  - Ekkor a leszármazottbeli konverizós operátor hívódik meg ősosztály pointerén keresztül elérve

Konverzió – többértelműség

- Ne írjuk meg mindkét konverziót, csak ha mindenképp szükséges
- Preferáljuk a konverziós konstruktort (más típusból a mi típusunkat).
  - A konverziós operátort pedig csak akkor, ha nem férünk hozzá a típushoz. Pl.:
    - Beépített típusra, pl. Complex-ből double-t
    - Ősből konverzió a leszármazottra és nem férünk hozzá a leszármazotthoz (a slicing ellentettje)
    - Egyéb

C++ stílusú típuskonverziók

- A nagyobb biztonságra való tekintettel a C++ saját konverziós szintaxist definiál
  - A C ugyanis jobban bízik a programozóban, a C++ viszont szigorúan típusos nyelv.
- A C megoldás egy kalap alá vesz bizonyos konverziós szándékokat
- Jobb lenne, ha pontosabban tudnánk megadni a konverzió célját. Ezeket segítik a következő operátorok

Statikus típuskonverzió

- static_cast < type-id > (kifejezés)
- Fordítás időben történik
- „biztonságos”, mivel tipikusan létezik konverziós út
- a konstansságot nem távolíthatja el
- Nem alkalmaz futási idejű típusellenőrzést
  - nem garantálja, hogy a konverzió eredményeképp a célobjektum teljes lesz

Újraértelmező típuskonverzió

- reinterpret_cast < type-id > (kifejezés)
- Mint a C-s elődje: a pointer típusát változtatja
  - vagyis azt, hogy milyen műveletek értelmezhetők egy memóriaterületen
- megengedi az egész típusok és a pointerek közötti konverziókat is

Dinamikus típuskonverzió

- dynamic_cast < type-id > (kifejezés)
- a hierarchián felfele, illetve lefele történő konverziókhoz szükséges
- Futásidőben történik
- Az osztályoknak polimorfaknak kell lenniük (legyen virtuális függvénye), különben ősről-leszármazottra konvertálás nem működik
- Használatához a futásidejű **típusinformációk kezelését be kell kapcsolnunk** a fordításkor
- A keresztbe konverziót is megoldja többszörös öröklés esetén, vagyis nagyon biztonságos.

Konstans típuskonverzió

- const_cast < type-id > (kifejezés)
- Képes konstans típust nem konstanssá tenni.
  - Ugyanis ez egy olyan veszélyes művelet, amelyet külön át kell gondolni

Kivételkezelés

Hibakezelés eddig

main

Jatek

JatekVege

RekordSzamitas

LemezreIr

A megoldás hátrányai

- körülményes
- keveredik a funkcionalitás és a hibakezelést megvalósító kód
- lassú lehet, amíg a hiba eljut a hívási fában a megfelelő hibakezelőhöz

Kivételek

- Exception
- Egy olyan mechanizmus, ami biztosítja, hogy ha hiba keletkezett valahol, akkor a futás (azonnal) a hibakezelőre ugorjon
- Példa

A mechanizmus

- A **try** egy **védett blokkot** jelent, amihez hibakezelő catch blokkok tartoznak.
- Ha valahol a hívási fában hibafeltételt találunk: **dobunk** egy kivételt a **throw** kulcsszóval.
- A throw-nak **paramétert** kell adni.
  - A throw paramétere lehet beépített típusú változó, de tetszőleges osztálybeli objektum is.

A mechanizmus

- A throw olyan, mint egy return,
  - de addig ugrik felfelé a fában, amíg egy megfelelő catch elkapja.
  - Vagyis addig, amíg egy olyan catch blokkot nem talál, aminek a paramétere kompatíbilis a dobott típussal
  - Ha van megfelelő catch blokk, annak paraméterébe beíródik a throw által dobott paraméter, ezt a catch blokkon belül fel tudjuk használni.
- A catch csak akkor fut le, ha hiba történt.
- A catch-ből kilépés után a try-catch blokk utántól folytatódik a végrehajtás, nem lép vissza a throw utáni utasításra.

Egymásba ágyazás

- A try-catch blokkok egymásba ágyazhatók
- Példa

Hiba újradobása

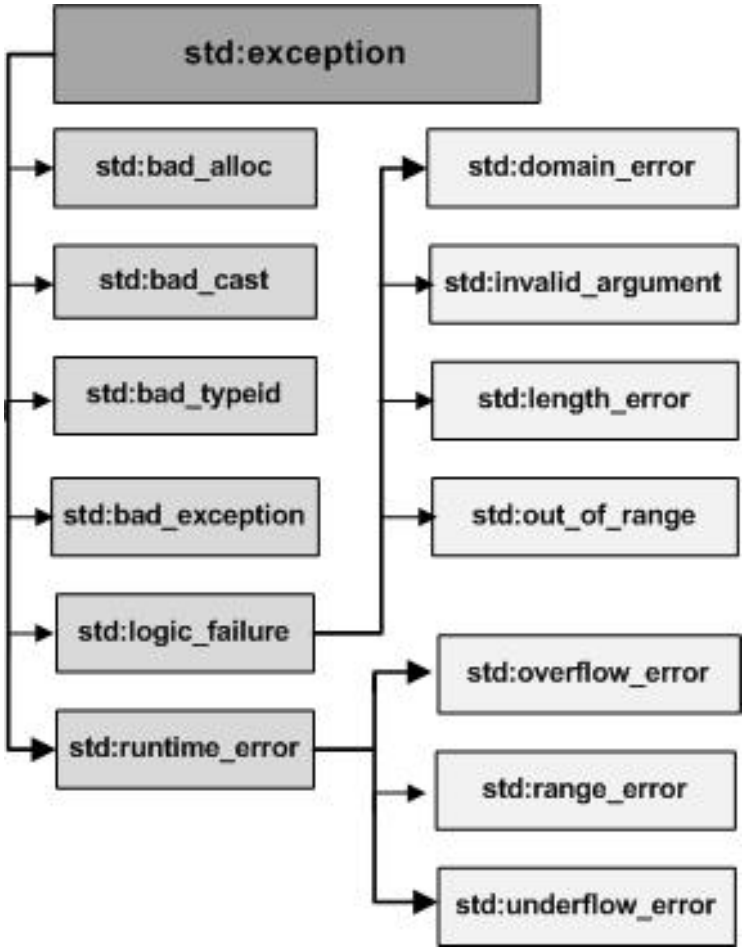
- Csak catch blokkon belül lehet
- Itt nincs paramétere a thrownak
- Ez az aktuális kivételt újradobja

- Példa

Típusosság

- Valójában nem szoktunk int-eket és char*-t dobni kivételként
  - hanem olyan objektumokat, amelyek valamilyen leírását tartalmazzák a hibának
- a Standard C++ library-ban vannak előre definiált szabványos exception típusok (hierarchiában)
- ezeket a kivételeket használjuk mi is
  - illetve ha saját kivétel típust szeretnénk létrehozni, akkor ezekből származtassunk le egy saját osztályt

Standard kódkönyvtár hierarchia



Kivétel elkapása

- Típus szerint lehet elkapni őket
  - ez szűrésre használható
- Több catch blokkot is meg lehet adni egymás alatt
  - ha a try blokkban kivételt dobott egy függvény a throw-val, akkor *sorrendben* végignézi a catch blokkokat és egy catch blokknál megáll, ha:

…megáll, ha:

1. catch(...)-ot talál
   - Ez mindent megfog, de nem kapjuk meg paraméterben az exception objektumot
2. típusa pontosan egyezik a kivétel objektum típusával
   - pl. int, string, exception
   - Ekkor másolat készül az eredeti exception objektumról, meghívódik a másoló konstruktora.
   - Olyan mintha függvény paraméter átadás lenne.

…megáll, ha:

3. Referencia vagy const referencia ugyanarra a típusra
   - catch(*exception*& e) vagy catch(const *exception*& e)
   - Mindig másolat készül a kivételről, nem a lokális objektumra kapnánk tehát referenciát
   - Ezt fogjuk szeretni.
4. A típus alaposztálya
   - catch(*exception* e), ahol a throw *spec_exception* volt, ahol a *spec_exception* az *exception* leszármazottja
   - Másolat készül.
5. Referencia vagy const referencia az osztály ősére
   - catch(*exception* & e) vagy catch(const *exception*& e).
   - Ezt is fogjuk szeretni.

…megáll, ha:

6. Pointerként, amire van konverzió a standard pointer konverziós szabályok szerint
   - Pl. catch(*exception** e) , akár ha leszármazottat dobtunk: throw new *spec_exception*;

- Az exception-kezelés egyik sarokköve, hogy ősosztálybeli típuson keresztül leszármazottat is meg tud fogni

Pointerdobással vigyázni!

```
void someFunction()
{
  exception ex;     //lokális objektum!
  throw &ex;
}
```

- Jobbnak tűnik:

```
void someFunction()
{
        throw new exception;
        // pointert dobunk egy új heap-beli objektumra
}
```

- De:
  - Maga a new is dobhat kivételt
  - Amikor elkapjuk, esetleg nem tudjuk, hogy a pointer a new-val lett létrehozva: ekkor bajban vagyunk, mert nem tudjuk, hogy meg kell-e hívni a delete-et

A stack visszacsévélése

- amíg a catch el nem kapja a kivételt, addig az összes (stack-en lefoglalt) lokális objektumot visszafelé haladva felszabadítja
  - a heap-en, new-val foglaltakat nem
  - természetesen meghívódnak az objektumok destruktorai.
- A destruktorban soha ne dobjunk kivételt,
  - mert ha aktív kivétel közben újabb kivételt dobunk, akkor azt nem tudjuk kezelni:
    - meghívódik a terminate, alapértelmezésben kilép az alkalmazás.
  - Ha mégis dobnánk, kapjuk is el a destruktorban és kezeljük le. Ekkor OK.

Kivételek összefoglalás

- Támaszkodjunk a C++ saját hibakezelő mechanizmusára
  - A helyén kezelni a hibát
  - Függetlenül az alkalmazás üzleti logikájától
- Használjuk a standard osztálykönyvtár exception alaposztályait