C++ extra óra

Kivételek kezelése

Mik azk a hibák amiket kezelni kell vagy érdemes?

1. programozói hiba (pl túlindexelés)

   Javasolt: Lelőni a programot. Mert ha észrevesszük hogy hibás, akkor bármi megtörténhet (pl. lehet hogy fájlokat szétszedi amiben dolgozik).

   Erre megoldás például az `std::terminate`. Ezt ha meghívjuk, a programot kilövi. Van más is, például az `exit` és `quick_exit`. A quick exit a destruktorokat se engedi lefutni.

2. Ha a fájl-al van probléma, akkor vagy olyan jellegű, hogy van esély recovery-re, akkor kommunikálhatunk is a programon keresztül. Például ha nem találja a fájlt, akkor azt lehet kezelni a program terminálása nélkül is.

3. Van, mikor lehet kezelni a problémát.

   Például ha van egy applikáció, ami magas szintű, mondjuk banki tranzakciókat végez, és egy alsó szintű mondjuk fájlkezelő libraryben van hiba, akkor azt valahogy vissza kell vinni az app-ba.

Esetleges megoldás:

```
int f (...)
{
        //...
        return -1;
        //..
}
```

Gond: Nem mindig tudunk egy nevezetes értéket visszaadni. De ha mondjuk pointert kell visszadni, akkor ez még annyira se működne.
Kérdés, egy alacsony szintről hogyan juttatjuk el a problémát magasabb szintre?

```
int e = g(...);
if (e)
        return e;
```

Ha valaki nem kezeli ebben a hibákat, akkor nem fog máködni az egész. Fentebb e-ben tároljuk el g-nek az eredményét, és utána e-től függ hogy az mit ad tovább. Probléma, hogy ha hiba van fent, akkor nagyon alacsony szintre kell visszaolvasni hogy a hiba forrását fel lehessen lelni.
Konkúzió: Visszatérési értékkel kivételeket kezelni nem nagyon éri meg.
C++ kód hibakezelése:

```
f()
{
        g();
        h();
        l();
}
```

Csak akkor írok hibakezelést, ahol valóban kell. Nem kell kilométernyi if, stb. Mivel sok if nincs, így ez még hatékonyabb is. Ráadásul elértük a célt: mert ez valóban a tetejére fog mászni.
Exception kezelés

```
void l()
{
        if (/* ... */)
        {
                throw e;
        }
}
```

Így jelzünk hibát

```
void a()
{
        try
        {
                f();
        }catch(E &e){
                //...
        }
}
```

`a`-ban van csak valami, ami lekezeli. Így `l`-ben dobja a hibát, átmegy `f`-en, és ott landol. (Ez a stack unwind, vagy stack visszatekerése)

```
struct S
{
        ~S()
        {
                std::cout << "dtor␣S" << std::endl;
        }
};

int main()
{
        S s;
        throw 5;
}
```

Mivel dobtam valamit amit sose kaptam el, ami nem definiált viselkedés.

```
struct S
{
        ~S()
        {
                std::cout << "dtor␣S" << std::endl;
        }
};

int main()
{
        try
        {
                S s;
                throw 5;
        }catch(int i){
                std::cout << "catch␣int" << std::endl;
        }
}
```

Lekezelt hibák esetén a destruktorok lefutnak rendesen.

```
int main()
{
        try
        {
                int *p = new int{5};
                f();
                delete p;
        }catch(int i){
                std::cout << "catch␣int" << std::endl;
        }
}
```

Ez memory leakhez vezethet, mert ha az `f()` dob kivételt, akkor a `delete p;` sor sose fog lefutni. Erre jó nekünk a RAII.

```
int main()
{
        try
        {
                std::unique_ptr<int> p(new int{5});
                f();
                delete p;
        }catch(int i){
                std::cout << "catch int" << std::endl;
        }
}
```

Itt már nincs probléma, mert `p`-nek a destruktora le fog futni. Ha exception safe kódot akarunk írni, akkor mindnen egyes alkalommal amikor mi lefoglalunk valamit, nekünk akkor biztosítani kell azt hogy azt az erőforrást felfogjuk szabadítani. Mert ha nincs egyetlen stacken lévő objektum se, ami oda mutatna, akkor resource leak következhet be.
Mit jelent az hogy exception safe?

1. weak exception guarantee:

   Akárhogyan is dobódik a programban a kivétel, sose fog reasource leaket okozni.

   Ezt mindig minden programnak tudnia kell.

   Ez egyszerűen megoldható, akkor a RAII-t kell hazsnálni: minden objektumot úgy hozunk létre, hogy destruktorelintézze ezt.

2. strong:

   tranzakcionális, azaz 2 dolog lehetséges `f` meghívásakor. Első lehetőség hogy lefut jól, és nincs semmi baj, a második esetben dobódik hiba, a program ennek hatására olyan állapotba jut, mintha f sose lett volna meghívódva. (azaz nem fordulhat elő az, hogy f-et kidobjuk, de a fele munkát már elvégzi.)

   CAS - Copy And Swap

   ```
   f (T &t)
   {
           T t2 = t;
   }
   ```

   Itt ha a copy konstruktor dob egy kivételt, akkor még a strong teljesül.

   ```
   f (T &t)
   {
           T t2 = t;
           g(t2);
           swap(t2, t);
   }
   ```

A swapot mindig lehet úgy megvalósítani, hogy nem dob kivételt. A swap az biztosan sikerült. Ha a copy konstruktor nem dobott semmit, akkor nem lesz gond. Így elértük a strong-ot, de ezzel egy extra másolást raktunk bele.

```
int g();

void f(std::unique_ptr<int> p, int);

int main()
{
        try
        {
                f(std::unique_ptr<int>(new int{5}, g());
```

```
        }catch(int i){
                std::cout << "catch int" << std::endl;
        }
}
```

Ez expection safe lesz? A válasz az hogy nem, mert a c++ban nem specifikált a végrehajtási sorrend. Előbb hajtódik végre a két paraméter kiértékelése mint az `f` meghívása. a paraméterek kiértékelési sorrend azonban már kérdéses. Az is lehet hogy létrehoz egy 5-öt a heap-en, és utána hoz neki létre egy unique pointert, és lehet g csak utána fut le. Ha g az dob egy kivételt.
Pontosabban:
Egy változó élettartama a konstruktor hívás végétől a destruktor hívás elejéig tart. Ha a konstruktor nem tud teljesen lefutni, a destruktor se fog. Itt előfordulhat hogy lefoglalja a memóriaterületet, meghívja g-t, és utána hívja meg a `unique_ptr` konstruktorát, de hag dob kivételt, akkor memory leak lesz.
A megoldás a szekvenciaponttal történő elválasztás, külön változóban hozom létre.

```
std::unique_ptr<int> p(new int{5});
f(std::move(p), g());
```

Ez már teljesen biztonságos lesz.
Problémát a konstruktorokkal

```
struct Y
{
        Y()
        {
                throw 5;
        }
}
struct X
{
        X() : y() {}

        Y y;
}
```

Ez nem lesz jó, hogyan kapjuk el? így:

```
struct Y
{
        Y()
        {
        throw 5;
        }
};
struct X
{
        X() try : y()
        {
                //constructor code
        }catch(int i)
        {
                std::cout << "X ctor exc" << std::endl;
        }

        Y y;
};

int main()
{
        try
        {
                X x;
```

```
        }catch (int i)
        {
                std::cout << "catch int main" << std::endl;
        }
}
```

Ekkor a kimenetben mindkét string benne lesz. Ha `X x` helyett `throw 5;`-t írnánk, és catch-be írok egy `throw`-t, akkor hajíthatjuk a hibát tovább. Ha ez lenne:

```
int main()
{
        try
        {
                throw 5;
        }catch (int i)
        {
                std::cout << "catch int main" << std::endl;
                throw 5;
        }
}
```

Ekkor csak az egyik dobódik el, a másik nem definiált.
Ha a fentebbi kdot nézzük, X konstruktorában elkapunk egy hibát. De semmit se tudunk tenni, mert az X nem lett létrehozva, és destruktora se fog lefutni, így kb csak logolni tudunk. Ilyenkor érdemes továbbdobni, és ha nem is tesszük, a ordító automatikusan odaír még egy `throw;`-t.
Sima `throw;`-t írok ,akkor azt dobom tovább amit elkaptam (de ez nyilván csak `catch`-ben működik.)
Itt ezen a `catch`-en belül számos egyéb probléma van: Mivel X nem lett létrehozva, ezért az adattagjaihoz nem férünk hozzá, így csak a globálisokhoz (amiket alapból nem is szeretünk) és a paraméterekhez férünk hozzá.
`new` ad erős garanciát.
A `new` általában elég bajos. Például, a memóriaallokáció dobhat-e exceptiont? Minek szabad, miben nem szabad használni?
Ökölszabály: destruktorból nem szivároghat ki exception. Egy hívásban egyszerre csak egy exception lehet.
Nézzük ezt meg:

```
struct Y
{
        ~Y()
        {
                std::cout << "dtor y" << std::endl;
        }
};



int main()
{
        try
        {
                Y y1;
                Y y2;
        }catch (int i)
        {
                std::cout << "catch int main" << std::endl;
        }
}
```

Kimenet ekkor:

```
dtor y
dtor y
```

```
catch int main
```

Mivan, ha kivétel az X-nek a destruktorából jön?

```
struct Y
{
        ~Y()
        {
                std::cout << "dtor y" << std::endl;
        }

};
struct X
{
        ~X()
        {
                std::cout << "dtor x" << std::endl;
                throw 5;
        }
};



int main()
{
        try
        {
                X x1;
                X x2;
        }catch (int i)
        {
                std::cout << "catch int main" << std::endl;
        }
}
```

Itt a RAII miatt mind `x1`-nek, mind `x2`-nek le fog futni a destruktora, viszont 2 különböző exception lesz hajítva, ami hibára vezet.
Destruktorban dobni szabad, de akkor destruktoron belül le kell kezelni. Ez egy **nagyon** nem definiált viselkedés. Itt GARANTÁLT hogy valami rossz működés fog történni. Nem definiáltnál még lehet hogy jól lefut a program, de ennél az esetnél garantált a hiba. Hasonló módon swap-nak se szabad dobnia.
Hogyan viselkedik az eldobott exception?

```
int main()
{
        try
        {
                char c = 5;
                throw c;
        }catch(int i){
                std::cout << "catch int main" << std::endl;
        }
}
```

A karakter implicit módon át tud konvertálódni integerré, de viszont ha változóval dobom át, akkor az implicit konverziókat nem vesszük figyelembe. A catch ádban semmiféle konverzió nem történhet. Lambdát semmi se tudja elkapni.

```
int main()
{
        throw
        {
                throw new int{5};
```

```
        }
        catch(int i)
        {
                std::cout << "catch int main" << std::endl;
                delete i;
                throw;
        }
}
```

MINDIG ÉRTÉK szerint dobunk, és catchelni néha referencia szerint. Itt fent kinek a dolga `i` törlése? Mival ha továbbdobjuk?

```
struct FileError
{
        virtual int errorCode() {return 0;}
};

struct FileNotExist : FileError
{
        virtual int errorCode() {return 1;}
};

int main()
{
        try
        {
                throw FileNotExist();
        }catch (FileError e)
        {
                std::cout << "catch main" << e.errorCode() << std::endl;
        }
}
```

Ez itt oké lesz, mert az ősosztály a leszármazottakat elkaphatja. Itt 0-t kapunk, mert slicing következik be. Ha referencia szerint veszünk át, akkor lesz 1.
Itt most a probléma az lesz az első esetben, hogy a `FileNotExist` statikus és dinamikus része is `FileError` típusú lesz. Mivel a `FileError` copy-konstruktora volt meghívva, enek megfelelően hozza létre. Ha viszont referencia szerint veszünk át, akkor a statikus része ennek `FileError` típusú lesz, de a dinamikus része (azaz amilyen memóriaterületen van) az `FileNotExist` típusú lesz.
Több catch Lehet több catch-el is játszani.

```
struct FileError
{
        virtual int errorCode() {return 0;}
};

struct FileNotExist : FileError
{
        virtual int errorCode() {return 1;}
};

int main()
{
        try
        {
                throw FileNotExist();
        }catch (FileNotExist e)
        {
                std::cout << "catch main" << e.errorCode() << std::endl;
                throw 5;
```

```
        }catch (FileError e)
        {
                std::cout << "catch main" << e.errorCode() << std::endl;
        }catch (...)
        {
                //...
        }
}
```

Itt az elsőbe fog belefutni. A legleszármazottabbat érdemes előreírni, és megyünk így kijjebb. a Z`catch(...)` mindenre illeszkedik. Mindig csak egy `catch` futhat le.

C++11
Ha van egy végrehajtási szál, melyben `f()` meghívja `g()`-t, ami meghívja `h()`-t, akkor ez egy úgynevezett THREAD. Ha több ilyenem is van, pl.
`f() -> g() -> h()`
`l() -> m() -> n()`
Itt a két thread dobhat két exceptiont, de threadenként csak 1-et.

C++11-ben jött egy úgynevezett `noexcept` dolog. Ez arra hogy, hogy megtiltja egy függvénynek hogy exceptiont dobjon. Ha egy noexcept függvényből hiba dobódik, akkor azonnal leáll a program. Egy ilyen függvényből SOHA nem fog kiszivárogni kivétel, vagy ha mégis, akkor le fog élni a program.
Érdemes egy függvényt `noexcept`-nek jelölni, ha nincs benne `noexcept`-et. Ezzel azonban óvatosan kell bánni, mert ha ne adj isten később még is kellene, és megváltoztatjuk `f()`-t úgy, hogy dobjon kivételt, akkor korábban destruktorokban problémákat okozhat. A `noexcept`-et komolyan át kell gondolni, ami egyszer az lett, maradjon is az.

Move szemantika:
Tegyük fel hogy `A+B+C+D`-ben minden egy nagy mátrix. A program kiszámolja `A+B`-t, utána `A+B+C`-t, utána meg `A+B+C+D`, amit utána átmásolok, vagyis így 3 nagy mátrixot kiszámoltam, ami nagy pazarlás.
Legyen úgy, hogy létrehozom `A+B`-t, és mivel ezt úgy is tönkretehetem majd, ezt módosítom amikor `C`-t hozzáadom, `D`-vel ugyanez, és ezt adjuk értékül.
Másolásnál ha hiba történik, nincs baj, mert az eredeti példány megmarad. Na de ezzel a move dologgal más a dolog, mert mondjuk egy nagy vector elemeinél valamelyik elem move-jánál exception dobódik, akkor valamelyik vektor a kettő közül az egyik elveszik. Ez baj, így próbáljuk meg visszarakni. De azonban ez is tönkremehet.
Így csak akkor move-olhatunk STL konténereket, ha az a move `noexcept`.