

# A programozás alapjai 2.

Hivatalos laborsegédlet hallgatók számára.

## Implicit, explicit konverzió

### Beépített típusok

C++-ban definiálva vannak standard konverziók a beépített, egymással kompatibilis típusokra. Kisebb méretű típusból nagyobb méretű típusra való konverzió probléma nélkül végrehajtható.

Pl.:

- `float` → `double`
- `unsigned` → `int`
- `short` → `int`
- `int` → `long`

Ilyen esetben három lehetőségünk van a konverzióra:

```
float f = 1.2;
double d1 = f; // implicit konverzió
double d2 = (double)f; // explicit konverzió
double d3 = double(f); // konstruktorral
```

Ellenkező irányba viszont, amikor nagyobb méretűből kisebb méretűre szeretnénk konvertálni, előfordulhat, hogy adatot veszünk. Ezt a compiler warning-gal jelzi (warning: possible loss of data), amit explicit konverzióval kerülhetünk el.

Pl.: `double`-ból `float`-ra konvertálva csökken a számábrázolás pontossága, warning elkerülése:

```
double d = 3.14;
float f = (float)d;
```

Bizonyos esetekben pedig nem is biztos, hogy ugyanazt az értéket kapjuk vissza konverzió után, ezekre nagyon figyeljünk:

- Negatív `int` érték konvertálása `unsigned int`-re:
  - általános szabály: `int` → `int MOD UINT_MAX+1`
  - `-1` → `unsigned int` által reprezentálható legnagyobb érték
  - `-2` → `unsigned int` által reprezentálható 2. legnagyobb érték
  - stb.
- A `bool` típus `false` értékének a 0 vagy null pointer felel meg, a `true` értéknek minden más
  - `bool` típusra történő konverzió ez alapján egyszerű
    - `0` → `false`
    - `null` → `false`
    - minden más → `true`

- bool típusból történő konverzió
  - false → 0
  - true → 1
- Lebegőpontos számról egész számra történő konverziókor a tört rész levágásával képződik az egész szám (nincs kerekítés alapértelmezetten, ehhez külön függvény kellene). Ha a keletkező egész szám nem ábrázolható a céltípusban (nem fér bele abba, amivé szeretnénk konvertálni), nem definiált, mi fog történni (undefined behavior).

## Pointerek

Null pointer bármilyen típusú pointer-ré konvertálható, és bármilyen típusú pointer konvertálható void pointerre.

## Osztályok

Ősosztály típusú pointerrel mutathatunk egy leszármazott osztály típusú objektumra.

Pl. ha Vehicle őse a Car-nak:

```
Car car;  
Vehicle* vechicle = &car;
```

Leszármazott osztályra mutató pointer átkonvertálható az ősosztályára mutató pointerre (ld. heterogén kollekciók).

Leszármazott objektumnak nem adható értékül ősosztály típusú objektum, mert nem tudná mivel kiegészíteni, viszont fordítva működik:

```
Car car;  
Vehicle vechicle;  
car = vechicle; // nem jó  
car = (Car)vechicle; // nem jó  
vechicle = car // jó  
vechicle = (Vehicle)car // jó, ugyanaz
```

Két, nem kapcsolódó típusú objektum közötti konverzió nem hajtható végre. Pointereikkel lehet, de teljesen értelmetlen. Pl.:

```
Dog dog;  
Car car;  
car = (Car)dog; // hiba
```

```
Car* pCar;  
Dog* pDog = &dog;  
pCar = (Car*)pDog; // nincs hiba  
pCar->tankFuel(); // hiba, kutya típusú objektumra mutat valójában
```

## Konverziós konstruktor

Minden olyan konstruktor, ami előtt nem áll explicit kulcsszó (ld. később), és egy paraméterrel hívható. Lehet több paraméterű konstruktor is, de ekkor az elsőt kívül mindegyiknek kötelező rendelkeznie default paraméterrel.

```
class Complex {
```

```
double im;
double re;
public:
    Complex(double re, double im = 0.0);
};
// ...
Complex c1 = 5.5;
Complex c2(5.5); // ugyanaz, mint az előző
Complex c3(1.0, 1.0);
```

Ennél a példánál maradva, ha lenne olyan függvényünk, ami Complex objektumot vár, és double-t kap, az nem lenne gond, mivel át tudja konvertálni Complex objektummá. Viszont ha Complex referenciát vár, azzal gond van. Csak akkor hívható double értékkel, ha nem egyszerű Complex referenciát vár, hanem konstanst. Ilyen függvény double értékkel történő hívásakor létrejön egy temporális Complex objektum a double értéket átkonvertálva, és erre kap referenciát a függvény. Pl.:

```
void func1(Complex c) {...}
void func2(Complex& c) {...}
void func3(const Complex& c) {...}

func1(2.0); // jó
func2(2.0); // nem jó
func3(2.0); // jó
```

Ilyen jellegű konverzióknál csak egy lépés lehet automatikus, a többi explicit ki kell írni. Például van automatikus konverzió int-ről double-re, double-ről Complex-re, de int-ről double-re már nincs. Az int → double → Complex láncban csak az egyik automatikus. Pl.:

```
func1(1); // hiba
func1((double)1); // jó, csak a double -> Complex automatikus
```

Ha egy művelet végrehajtására több út is rendelkezésre áll, azt fogja választani, amihez nincs szükség konverzióra. Konverziót csak akkor használ, ha nem talál konverzió nélküli egyező módot.

Pl.:

```
func(double d) {...}
func(Complex c) {...}

func(1.0); // az első függvény hívódik meg
```

## explicit kulcsszó

Konverziós konstruktor elé explicit kulcsszót írva, az adott konverzió nem hajtható végre implicit módon, mindig explicit módon kell jelezni a konverziót.

```
class Complex {
    double im;
    double re;
public:
    explicit Complex(double re, double im = 0.0);
};

void func(Complex c) {...}

//...
Complex c(5.0); // jó
```

```
func(2.0); // hiba  
func((Complex)2.0); // jó
```

## Konverziós operátorok

A konverziós konstruktor arra volt jó, hogy valamilyen más típusból hozzunk létre sajátot. A konverziós operátor pedig pont ennek az ellenkezőjét fogja csinálni, tehát saját típusból hoz létre egy másikat.

```
class Complex {  
    double im;  
    double re;  
public:  
    Complex(double re, double im = 0.0);  
    operator const double() const {return re;}  
};  
// ...  
Complex c(2.5, 3.0);  
double d = c;
```

Erre is igaz, hogy automatikusan, implicit módon csak egy lépést tud végrehajtani, a többi lépést explicit cast-olással kell.

```
Complex c(2.5, 3.5);  
float f1 = c; // nem jó  
float f2 = (double)c; // jó  
float f3 = (float)(double)c; // jó
```

A konverziós operátor öröklődik, tehát a leszármazottban is használható felüldefiniálás nélkül is.

```
class A : public Complex {...}  
A a(2.0, 5.0);  
double d = a; // jó
```

Lehet virtuális az őosztályban, és a leszármazott osztályban felüldefiniálhatjuk.

Próbáljunk meg inkább konverziós konstruktort használni. Ha nagyon muszáj konverziós operátort, akkor figyeljünk a többértelműség elkerülésére (többféleképp is végre lehet hajtani ugyanazt).

## Összefoglalás

```
class Complex {  
    double im;  
    double re;  
public:  
    Complex(double re, double im = 0.0);  
    Complex& operator=(const double& d);  
    operator double() const;  
};  
// ...  
double d = 2.5;  
Complex c = d; // konverziós konstruktor  
c = d; // = operátor  
d = c; // konverziós operátor
```

## C++ saját típuskonverziói

A C++ saját konverziós szintaxisai, melyek nagyobb biztonságot nyújtanak, használatukkal megadható a konverzió célja is.

### **static\_cast<target\_type>(expression)**

Nem alkalmaz futás idejű típusellenőrzést, nem garantálja, hogy a konverzió eredményeképp a célobjektum teljes lesz. Pl. ha leszármazott osztállyá konvertálunk egy ősz osztály típusú objektumot, attól még a leszármazott osztály plusz függvényei nem lesznek hívhatók. Nem távolítja el a konstans tulajdonságot.

### **reinterpret\_cast<target\_type>(expression)**

Bármely pointer típust képes átkonvertálni bármely más pointer típusná, még teljesen független típusná is. Nem ellenőrzi sem a pointerek típusát, sem a pointerek által mutatott objektumok típusát. Nem távolítja el a konstans tulajdonságot. Megengedi az egész típusok és pointerek közti konverziót, illetve pointerek helyett használható referencia is.

Annyit garantál és semmi többet, hogy ha egy A\* típusú pointert castolunk B\* típusúvá (A és B helyén bármilyen típus szerepelhet), majd az eredményt visszacastoljuk A\*-gá, akkor ennek az eredménye egy érvényes A\* típusú pointer lesz.

### **dynamic\_cast<target\_type>(expression)**

Az öröklési hierarchián felfele illetve lefele történő cast-oláshoz használható polimorf típusok között. Privát öröklés esetén felfele nem használható, futás idejű hibát kapunk. Nagyon biztonságos, biztosítja, hogy az eredményként kapott objektum teljes és érvényes legyen. Ha nem sikerül a cast-olás, null pointert ad vissza, ezért az eredményét mindig ellenőrizni kell használat előtt. Nem távolítja el a konstans tulajdonságot.

```
Base * pBase1 = new Derived;
Base * pBase2 = new Base;
Derived * pDerived;
pDerived = dynamic_cast<Derived*>(pBase1); // sikeres, nem null
pDerived = dynamic_cast<Derived*>(pBase2); // nem sikeres, null
```

### **const\_cast<target\_type>(expression)**

Az egyedüli cast-olási mód, mely képes konstans típust nem konstanssá konvertálni. Csak erre képes, de mivel veszélyes művelet, külön jelölést kapott. Ha az így kapott, már nem konstans objektumot nem módosítjuk, nincs gond. Viszont ha módosítjuk, nem definiált a további működés (undefined behavior).

## Kivételkezelés

### 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 szíten el kell végezni, ami ráadásul teljesen felesleges. A különbséget az alsó táblázatban látható példakódok szemléltetik:

Exception	Error code
<pre> void f1() {     try {         // ...         f2();         // ...     } catch (const some_exception&amp; e) {         // ...hiba kezelése...     } }  void f2() { ...; f3(); ...; } void f3() { ...; f4(); ...; } void f4() { ...; f5(); ...; } void f5() {     // ...     if (/*...hiba feltétel...*/)         throw some_exception();     // ... } </pre>	<pre> 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() {     // ... } </pre>

```
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**-cselel jelt 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 helyezünk 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 őséire. 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::exception`-ből örököltetni, hiszen ha valaki olyan try-catchet ír, ami `std::exception`-t kap el, jó okkal feltételezhetjük, hogy a mi custom exceptionünket is el szeretné kapni. Egyébként nem érdemes konkrétan `std::exception`-t 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 hibaszpecifikusabb 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 egészszel, 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.