Natív Programozás

Dr. Siket István

Szegedi Tudományegyetem
Informatikai Intézet
Szoftverfejlesztés Tanszék

2025

Tartalom

Tartalom

Előfeltételek, jelentkezés, követelmények

- Előfeltételek
  - Programozás I. gyakorlat vagy Objektum-orientált Programozás gyakorlat
    - Aki nem teljesítette, **NE** vegye fel a kurzust!
- Kurzusra jelentkezés
  - A kurzusra a Neptunon keresztül lehet jelentkezni
  - Kurzusfelvevő lap
    - Indokolt esetben
    - Csak olyan gyakorlatra lehet jelentkezni, ahol van hely
- **A követelmények a CooSpace-ben lesznek publikálva**

Gyakorlat anyaga, gyakorlás

- Gyakorlat anyaga
  - `https://okt.inf.szte.hu/native/`
- Minden gyakorlaton ugyanaz az anyag lesz
  - Az adott heti feladat megoldását felrakjuk
  - Az előző hetit folytatjuk
- Gyakorló feladatok
  - biro2-n elérhetőek lesznek

Számonkérés, ZH

- A félév során 3 ZH
  - Saját gyakorlaton kell megírni
    - A gyakorlat időpontjában és termében
  - Gép előtt, bíró értékeli ki a megoldást
  - 90 pontot lehet szerezni
    - **Legalább 50%-ot el kell érni a gyakorlat teljesítéséhez**
  - Igazolatlan hiányzás esetén a gyakorlat **nem értékelhető**
    - Igazolatlan hiányzás esetén pótlásra nincs lehetőség
- Általunk biztosított segédanyag használható
  - Egyéb segítség (telefon, okosóra, AI) nem használható
  - Nem megengedett segítség használata esetén jegyzőkönyvet készítünk
    - **A ZH nem folytatható és nem értékelhető**
- ZH időpontok, pontszámok
  - 5. gyakorlat (okt. 6., 8., 9., 10., 14.), 30 pont
  - 9. gyakorlat (nov. 3,. 5., 11., 13., 14.), 35 pont
  - Utolsó hét (dec. 8-12), 25 pont
- Technikai probléma
  - Gép lefagyása (vagy hasonló gondok) esetén 10 perc plusz idő

Gyakorlati jegy meghatározása

- Az elégséges jegyhez a követelmények
  - **ZH-kból legalább 50%-ot el kell érni**
  - **Minimum 50 pontot kell szerezni a ZH-kból és házikból**
- Plusz pontokat lehet órai munkával szerezni
  - Nem számítanak bele a minimum követelménybe
- Ponthatárok

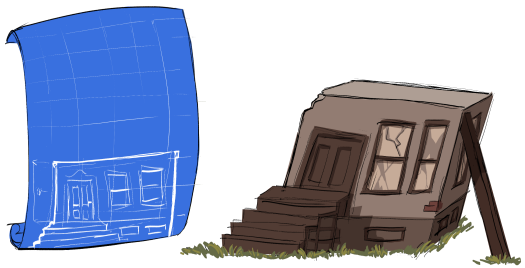
89	-		jeles (5)
76	-	88	jó (4)
63	-	75	közepes (3)
50	-	62	elégséges (2)
0	-	49	elégtelen (1)

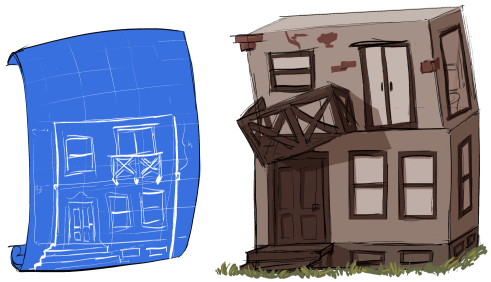
Gyakorlat pótlása, javítási lehetőség a szorgalmi időszakban

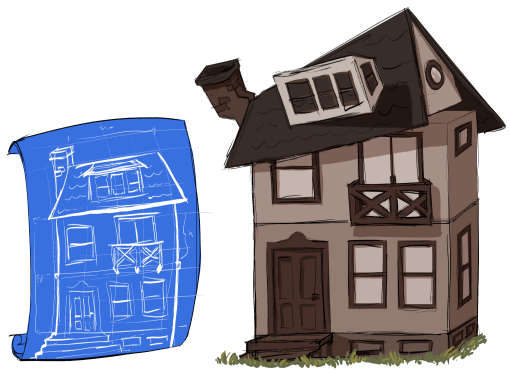
- Igazolatlan hiányzás esetén nem lehet pótolni
  - A félév nem értékelhető
- Igazolt hiányzás esetében a ZH pótolható
  - 1. ZH esetében: okt. 10. péntek (korlátozott), okt. 17. péntek
  - 2. ZH esetében: nov. 14. péntek (korlátozott), nov. 21. péntek
  - 3. ZH esetében: dec. 12. péntek (korlátozott), dec. 15. vagy 16.
    - Pontos időpont az utolsó héten lesz kihirdetve
- Javítási lehetőség a szorgalmi időszakban
  - Újra lehet írni valamelyik ZH-t a pótlás időpontjában
    - **Csak egyet lehet újraírni**
    - „Aki az adott ZH-t pótolja, az nem javíthat”
  - CooSpace-en kell jelentkezni

Gyakorlati jegy javítása a vizsgaidőszakban

- Elégtelen gyakorlati jegy esetén a vizsgaidőszakban lehet javítani
  - Neptunban kell jelentkezni
  - A teljes féléves anyagból egy ZH
    - A házi pontszámok és gyakorlati pluszpontok megmaradnak

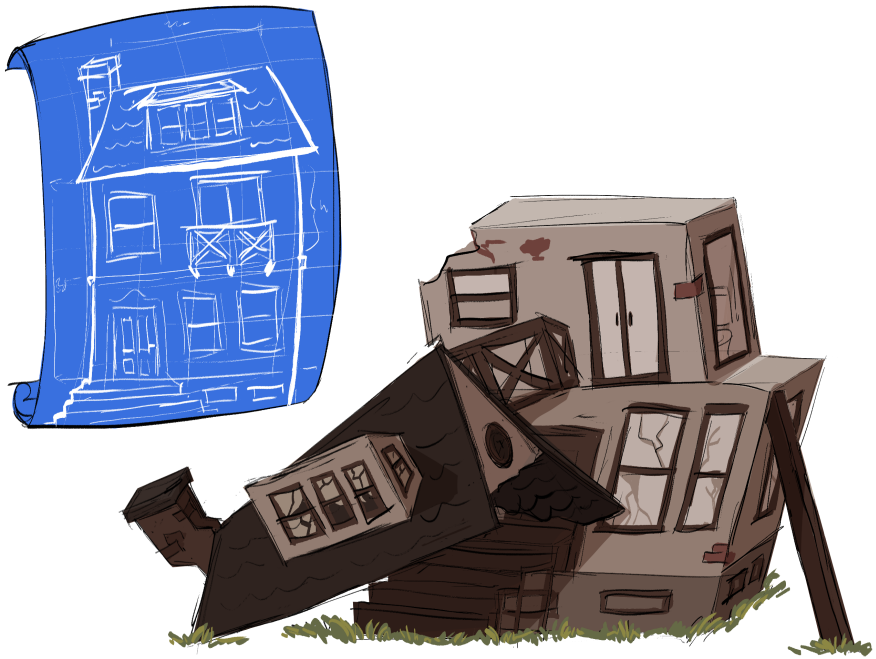






Gyakorlati jegy javítása a vizsgaidőszakban

- Elégtelen gyakorlati jegy esetén a vizsgaidőszakban lehet javítani
  - Neptunban kell jelentkezni
  - A teljes féléves anyagból egy ZH
    - A házi pontszámok és gyakorlati pluszpontok megmaradnak



Házi feladat

- Heti házi feladat
  - 10 pontot lehet szerezni, bírón kell beadni
  - Nem lehet pótolni
- Ha a kapcsolódó ZH pontszáma nem éri el a 40%-ot, a házi feladat pontszáma törlődik
  - Első 4 házi feladat esetében az 1. ZH-n kell elérni a 40%-ot
  - 5. , 6., és 7. házi feladat esetében a 2. ZH-n kell elérni a 40%-ot
  - 8. , 9., és 10. házi feladat esetében a 3. ZH-n kell elérni a 40%-ot

Előadás

- Vizsgának nem előfeltétele a gyakorlat teljesítése
- CooSpace teszt
- Ponthatárok

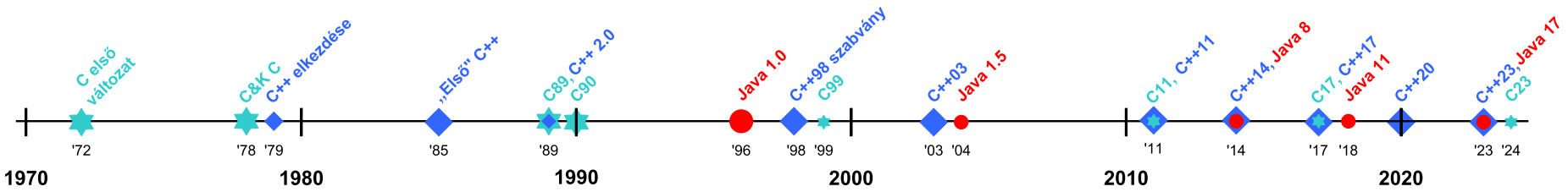
89	-	100	jeles (5)
76	-	87	jó (4)
63	-	75	közepes (3)
50	-	62	elégséges (2)
0	-	49	elégtelen (1)

- Megajánlott jegy
  - Jövő héten pontosítom ...

Tartalom

Tartalom

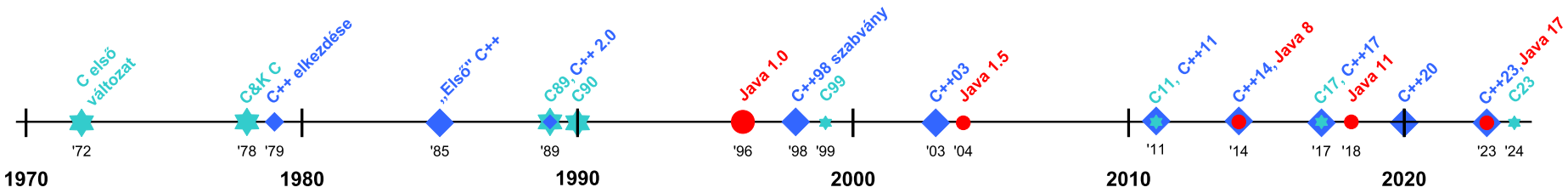
C nyelv



- 1972: C első változata
- 1978: C&K C
  Brian Kernighan, Dennis Ritchie
- 1989: C89, ANSI X3.159-1989
- 1990: C90, ISO/IEC 9899:1990
- 1999: C99, ISO/IEC 9899:1999
- 2011: C11, ISO/IEC 9899:2011
- 2017: C17, ISO/IEC 9899:2018
- 2024: C23, ISO/IEC 9899:2024

- Procedurális, imperatív
- „Alacsony szintű"
  - Gépközeli
- „A fejlesztőé a felelősség"
  - Hibakezelés
  - Memória felszabadítása

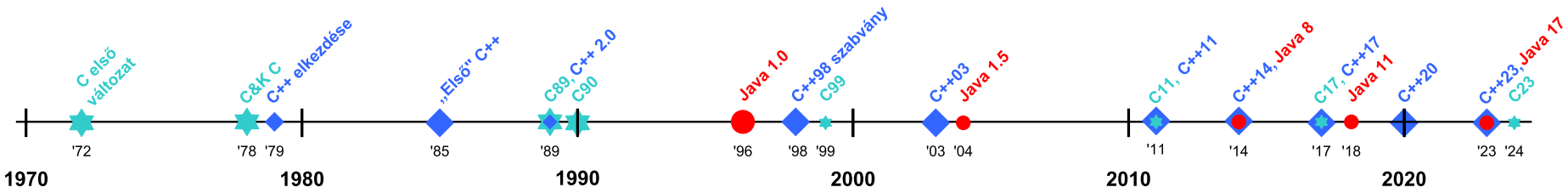
Java nyelv



- 1996: Java 1.0
- 2004: Java 1.5
- 2014: Java 8 (LTS)
- 2018: Java 11 (LTS)
- 2021: Java 17 (LTS)
- 2023: Java 21 (LTS)

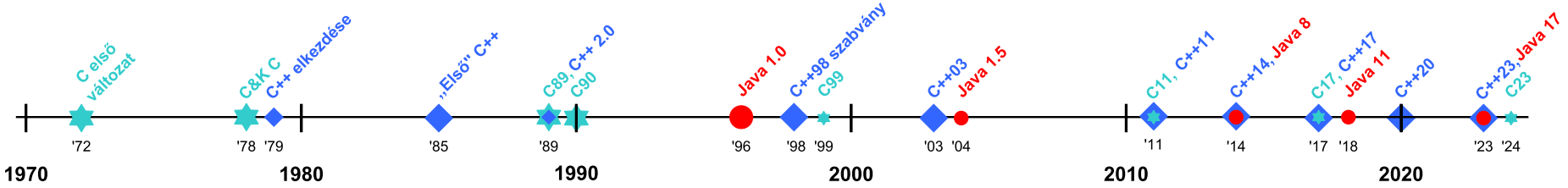
- Objektum-orientált
- „Magas szintű"
  - GUI, hálózat, ...
- Biztonságos (ellenőrzések)
  - Tömb túlindexelés
  - Inicializálás hiánya
  - Kivétel elkapása/jelzése
  - Memória felszabadítása

C++ története



- 1979: Bjarne Stroustrup elkezdte kidolgozni
  - A C nyelv továbbfejlesztése volt
  - „C with Classes" volt a neve
- 1983: C++ lett a neve
  - A ++ arra utal, hogy a C nyelvez képest sok új lehetőséggel bővült
- 1985: megjelent az „első változat"
  - Nem ISO szabvány
- 1989: megjelent a 2.0

C++ története - ISO/IEC szabványok



- 1998: ISO/IEC 14882:1998 (C++98)
- 2003: ISO/IEC 14882:2003 (C++03)
- 2011: ISO/IEC 14882:2011 (C++11, C++0x)
  - Bjarne Stroustrup: Surprisingly, C++11 feels like a new language: The pieces just fit together better than they used to and I find a higher-level style of programming more natural than before and as efficient as ever.
- 2014: ISO/IEC 14882:2014 (C++14, C++1y)
- 2017: ISO/IEC 14882:2017 (C++17, C++1z)
- 2020: ISO/IEC 14882:2020 (C++20, C++2a)
- 2023: C++23

C++ nyelv

Programozási paradigmák

- Procedurális (C nyelv)
  - C++-ban is lehet „így” programozni
    - Azonban még így is bővebb, mint a C
  - A különbségek ellenére a C++-t a „C folytatásának” tekintjük
- Objektum-orientált (Java)
  - Hasonló a Javahoz, de sok eltérés van
    - Javaban könnyebben és hatékonyabban (gyorsabban) lehet programozni
    - De például ha a performancia számít, akkor C++
- Generikus
  - Az algoritmusokban használt típusok a példányosításnál lesznek meghatározva
    - Például Java generikus programozás
  - Azonban a C++ itt is más, mint a Java
    - Fordítás idejű számítások

C és C++ nyelv kapcsolata

- A C++ a C továbbfejlesztése
  - Akkor a C nyelv a C++ nyelv része? **NEM**
- Minden C program lefordítható C++ programként is? **NEM**
  - A C++ kulcsszavak problémát okoznak
- Ha pont ugyanazt a kódot fordítjuk, akkor ugyanúgy fut? **NEM**
  - Például a karakterliterálokat ('a', 'b', ...) másképp kezelik
    - C-ben egész típus
    - C++-ban karakter típus

```
#include <stdio.h>

int main() {
  if (sizeof('a') == sizeof(int))
    printf("C program\n");
  else if (sizeof('a') == sizeof(char))
    printf("C++ program\n");
}
```

  - Vannak más apró eltérések is: `https://www.geeksforgeeks.org/write-c-program-produce-different-result-c/`

Hol tart most a C++?

- Különböző szempontok alapján lehet vizsgálni
  - Általában benne van a TOP10-ben
- Online ranglisták
  - `https://www.tiobe.com/tiobe-index`
  - `https://spectrum.ieee.org/top-programming-languages-2024`
  - `https://pypl.github.io/PYPL.html`

Előnyök és hátrányok

- Performancia
  - Natív kód, kevesebb ellenőrzés, stb.
- Szinte minden platform támogatott
  - Windows, Linux, iOS, Android, beágyazott rendszerek, ...
  - Hordozhatóság korlátozott
- USA kormánya: használjunk memória-biztonságos nyelveket
  - `https://www.infoworld.com/article/2336216/white-house-urges-developers-to-dump-c-and-c.html`

Felhasználási terület

- Operációs rendszer
  - Kernel, driver, ...
- Számítógépes játékok
  - Engine: Unity, Unreal, REDengine, ...
    - `https://www.mycplus.com/featured-articles/list-of-top-100-game-engines-written-in-c-c/`
  - Witcher 3, Counter-Strike, Doom III Engine, World of Warcraft
- Beágyazott rendszerek, IoT eszközök
  - Telefon, TV, óra, hűtő, ...
    - Frameware, alkalmazások, ...
- Számításigényes alkalmazások
  - Tudományos számítások, szimulátorok, ...
  - AI számításigényes része
- Webböngészők
  - Jobb performancia

Félév anyaga

- Típusok
- **Vezérlési szerkezetek**
- **Objektum-orientáltság**
  - **Osztály**
    - **Attribútum**, **metódus**
    - **Láthatóság**, friend
    - **this**, const, **static**
    - **Konstruktor**, destruktor
  - **Operáció kiterjesztés**
    - Operátor kiterjesztés
  - **Öröklődés**
    - **Felüldefiniálás**
    - **Polimorfizmus**
    - Többszörös öröklődés
  - **Objektumok élettartalma**
    - **Létrehozás**, megszűnés
    - **Stack**, **heap**, **static**, global
- Beolvasás, kiírás
- Típuskonverzió
  - C-szerű
  - C++ típuskonverzió
- Tárolók
  - vector, set, map, ...
  - Bejárások
    - **Iterator**, **lambda**
  - Algoritmusok
- **Kivételkezelés**
  - RAII
- **Névtér (csomag)**
- Dinamikus memóriahasználat
  - Objektum másolása, törlése
- Template osztályok
  - „Generikus” osztályok

Tartalom

Tartalom

Első C++ program

- Helló Világ!

```
#include <iostream>

int main() {
  std::cout << "Hello Vilag!" << std::endl;
  return 0;
}
```

- Üres program

```
int main() {
}
```

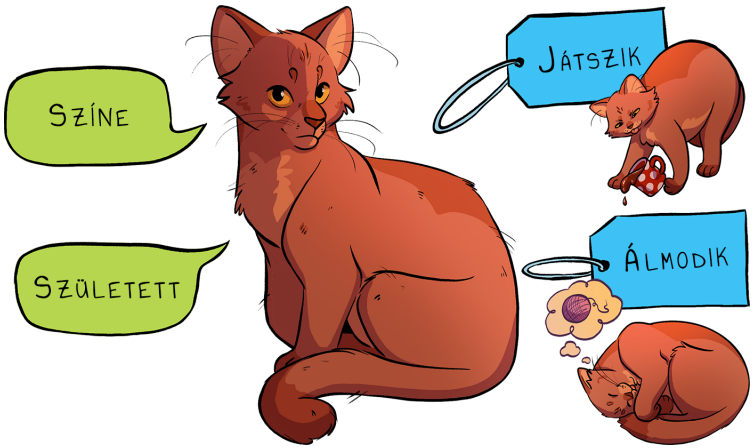
Program belépési pontja

Visszatérési érték

- Program belépési pontja a `main` függvény
  - `int main() { /* ... */ }`
  - `int main(int argc, char* argv[]) { /* ... */ }`
  - Implementáció-függő megvalósítás
    - `int main(int argc, char* argv[], char* env[]) {}`
- Visszatérési érték
  - 0 jelzi a helyes futást
  - Minden más érték hibát jelez
- Visszatérési érték megadása
  - Ha a program futása eléri a main függvény végét (`main`-ben található) `return` utasítás nélkül, akkor a program 0-val tér vissza
    - Nem kell kiírni a `return 0;`-t
    - Ennek ellenére érdemes kiírni
    - A diákon hely hiányában nem lesz kiírva

Forráskód szervezése

- Hogyan lehet, illik szervezni a kódot C++-ban?

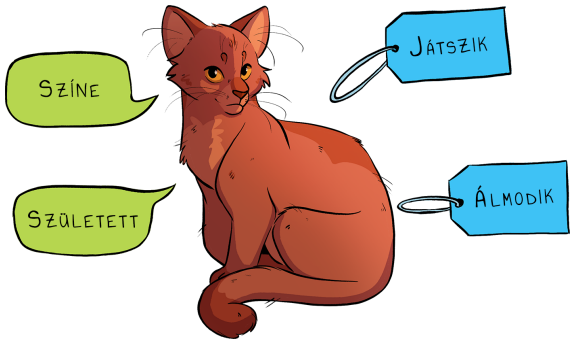


`macska.h` vagy `macska.cpp`

```
class Macska {
  string szin;
  int szul;

  void jatszik() {
    /* bogret lelok */
  }

  void almodik() {
    /* gombolyagrol almodik */
  }
};
```

Interfész és implementáció

- Interfész
  - Tulajdonságok
  - Mit csinál



macska.h

```
class Macska {
  string szin;
  int szul;

  /* A jatszik ... */
  void jatszik();

  /* Az almodik ... */
  void almodik();
};
```

- Implementáció
  - Hogyan

macska.cpp

```
#include "macska.h"

void Macska::jatszik() {
  /* ... */
}

void Macska::almodik() {
  /* ... */
}
```

Projekt felépítése

- Interfész
  - Header fájlok (`.h`, `.hxx`)
    - Egy header fájlban lehet több osztály is
  - Felhasználás esetén elég tudni, hogy *mit csinál*
- Implementáció
  - Forrás fájlok (`.cpp`, `.C`, `.cxx`, `.cc`, `.c++`)
  - Implementáció, ami „irreleváns”
- Gyakorlaton: minden egy fájlban

Fordítás (build-elés) lépései

- Preprocesszálás: forráskódból „forráskódot” készít
  - Preprocesszor direktívák kezelése
- Fordítás: forráskódból assembly kódra fordítás
- Assembly: assembly kód gépi kóddá alakítása
- Linkelés: bináris fájlok összeszerkesztése

Preprocesszálás

- Fontosabb feladatok
  - „Behelyettesítés”
    - Header fájl (`#include`)
    - Macro (`#define`)
  - Feltétel kiértékelése
    - `#if` FELTETEL
  - Kommentek eltávolítása
- `g++ -E hw.cpp -o hw.ii`
  - `-E`: preprocesszálás
  - `.ii`: preprocesszált fájl

Preprocesszálás

Header fájlok, `include`

- A header fájlok tartalmát „bemásolja” a `#include` helyére
  - A header fájlok is tartalmazhatnak további `#include` utasításokat
    - Rekurzívan lesznek feldolgozva és bemásolva
- „Többszörös bemásolás” gondot okozhat
  - Például egy típus def. többször szerepel
    - Megoldás: makró védőfeltétel (`pragma`)
    - Csak elsőre lesz „bemásolva”
- Header fájl helye
  - Standard header fájlok
    - `#include <iostream>`
    - Fordítóprogram tudja hol kell keresni
  - „Egyéb” header fájlok
    - Saját rendszerből vagy külső könyvtárból
    - Pl. `#include "macska.h"`
  - Keresési könyvtár megadása: `-I UTVONAL`
    - `g++ macska.cpp -I macska`

Preprocesszálás

Makró

- Definiálhatunk makrókat, amit a preprocesszor behelyettesít
  - Nem kell, hogy C++ szintaxis szerint „teljes” legyen
    - Az „elkészült” kódnak kell annak lennie

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

#define HW "Hello Vilag!"

int main() {
  cout << HW << endl;
}
```

```
/* ... */
using namespace std;



int main() {
  cout << "Hello Vilag!" << endl;
}
```

- „Előre definiált” makró

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

int main() {
  cout << INT_MAX << endl;
}
```

- Fordító által definiált

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

int main() {
  cout << __LINE__ << ". sor" << endl;
  cout << __LINE__ << ". sor" << endl;
}
```

Fordítás lépései

Object fájl

- Assembly kód
  - Gépi kódra fordítja a forrást
  - `g++ hv.cpp -S`
    - Kimenet: hv.s
- Bináris kód
  - Az assembly utasítások lefordítva
  - `g++ hv.cpp -c -o hv.o`

Fordítás lépései

Linkelés

- Hivatkozások feloldása
  - Fordításhoz elég, ha deklarálva van egy metódus, osztály, ...
    - Megmondom, hogy van ilyen, lehet használni
    - De nem mondom meg, hogy mi a megvalósítása
  - Futtatáshoz szükséges az implementáció is

`macska.h`

```
class Macska {
  // ...
};

void beolt(Macska& m);
```

`macska.cpp`

```
#include <macska.h>

void beolt(Macska& m) {
  // ...
}
```

`main.cpp`

```
#include <macska.h>

int main() {
  Macska macska;
  beolt(macska);
}
```

- One Definition Rule (ODR)
  - „Mindennek definiálva kell lennie pontosan egyszer”

Fordítóprogramok

- GCC (The GNU compiler collection)
  - `https://gcc.gnu.org/`
  - Windows (Cygwin, MinGW), Linux, Mac, ...
- Clang
  - `https://clang.llvm.org/`
  - Windows, Linux, Mac (XCode), ...
- Visual Studio
  - Windows

Build rendszerek

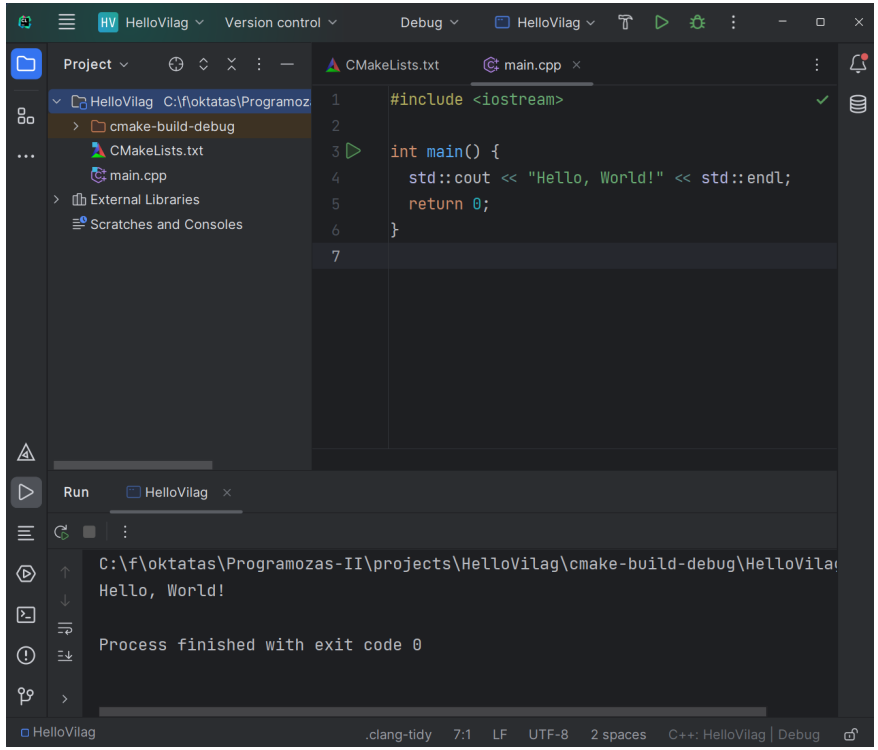
- „Kis rendszer” esetén lehet használni a fordítási parancsokat
  - Kevés fájl, kevés paraméter, ...
  - Nincs sok konfiguráció
- Nagy rendszerek build-elése
  - Sok fájl, paraméter, ...
  - Különböző konfigurációk lehetnek
    - „Mit fordítsak?”
    - Eltérő paraméterezés
- Build rendszerek
  - Leírjuk, hogyan kell build-elni a rendszert
    - Input fájlok, paraméterek, ...
    - Kimenet
  - make
    - `Makefile` tartalmazza a leírást
    - `make` program „értelmezi és végrehajtja”
  - VS projekt fájl
    - VS-ben meg lehet nyitni

cmake

- Build rendszerek hasznosak, de más-más leírást használnak
  - Mi van, ha két különböző eszközzel akarunk fordítani?
    - Linux alatt GCC
    - Windows alatt VS
- „Általánosan” definiáljuk a build-elést
  - Forrásfájlok, paraméterek, kimenet(ek), ...
  - `cmake` segítségével legenerálhatjuk a megfelelő projekt fájlt

Fejlesztőkörnyezet

- Előnyök
  - Segít a fejlesztés során
    - Hivatkozások
    - Kiegészítések
  - Build
    - Egyetlen gomb megnyomása
  - Hibák megtalálása
    - Kijelöli
    - Javaslatok
- Fejlesztőkörnyezetek
  - Kereskedelmi
    - Visual Studio, **CLion**
  - Ingyenes (nyílt forráskódú, platformfüggetlen)
    - Eclipse, Code::Blocks, CodeLite, Visual Studio Code, ...



Tartalom

C++ szabvány

- A C++ szabvány leírja a C++ nyelvet
  - Azonban számos esetben (szándékosan) nem definiál minden részletet
  - Definiálja, hogy az adott konstrukció nem definiált viselkedéshez vezet
- Különböző platformokon „másra” van szükség
  - Például a pointer vagy objektum mérete
    - 32 biten más méretre van szükség, mint 64 biten
- Optimalizálási lehetőség
  - A fordítóprogram könnyebben optimalizálhat
- Performancia
  - Például nincs tömb túlindexelés ellenőrzés
    - „Gyorsítás”, mert nem ellenőrzi
    - Veszélyes, mert túlindexelhetjük a tömböt
  - Nincs automatikus memóriafelszabadítás (heap)
    - A programnak nem kell foglalkoznia vele (gyorsabb)
    - Memória szivárgás

Implementáció-függő viselkedés

Implementation-defined behavior

- A program viselkedése **implementáció függő**,
  de **a megvalósításnak dokumentálva kell lennie**
- Példa: a `size_t` egy előjeltelen egész szám
  - `typedef /*implementation-defined*/ size_t;`
  - A `sizeof` operátor visszatérési típusa
  - Legalább 16 bit
- A szabvány nem specifikálja, hogy pontosan hány bájtos legyen
  - *A legnagyobb objektum méretét is tudni kell ábrázolni rajta*
- Például ha `unsigned int` típust használunk indexelésre
  - 64 bites rendszereken gond lehet, ha az index nagyobb, mint `UINT_MAX`

Nem specifikált viselkedés

Unspecified behavior

- A program viselkedése **implementáció függő**,
  de **a megvalósításnak NEM kell dokumentálva lennie**
  - Szabadság az optimalizációk számára
  - Bármely kimenet valid eredménynek tekinthető
- Ugyanaz a kód másik fordítóval fordítva más eredményt adhat
  - Más paraméterekkel fordítva is lehet eltérő a futási eredmény

Nem specifikált viselkedés

Példa: függvény paraméterlistájának kiértékelési sorrendje

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

int i = 9;

int foo() {
  cout << "foo ";
  return i;
}

int goo() {
  cout << "goo ";
  i = 6;
  return 0;
}

int add(int a, int b) {
  return a + b;
}

int main() {
  cout << add(foo(), goo()) << endl;
}
```

- Lehetséges kimenetek

GCC 12	`goo foo 6`
VS 2017	`goo foo 6`
Clang 12	`foo goo 9`

- A függvényhívásnál a paraméterlista kiértékelési sorrendje nem specifikált
  - Mindegyik kimenet helyes

Nem definiált viselkedés

Undefined behavior

- A szabvány nem definiálja, hogy mi legyen a program működése
  - **Bármilyen viselkedés lehetséges**
  - **A fordítóprogramnak nem feladata ezek felismerése**
  - Nem is célja, hogy elrontsa a programot, de ...
    - Például optimalizál és így „megváltozik az eredmény”
- Példa: előjeles egész szám túlcsordulása

```
int main() {
  int i = INT_MAX;
  cout << i+1 << endl;
}
```

  - Mi lesz a program futásának az eredménye?

Nem definiált viselkedés

Undefined behavior

- A szabvány nem definiálja, hogy mi legyen a program működése
  - **Bármilyen viselkedés lehetséges**
  - **A fordítóprogramnak nem feladata ezek felismerése**
  - Nem is célja, hogy elrontsa a programot, de ...
    - Például optimalizál és így „megváltozik az eredmény”
- Példa: előjeles egész szám túlcsordulása

```
int main() {
  int i = INT_MAX;
  cout << i+1 << endl;
}
```

  - Mi lesz a program futásának az eredménye?
    - -2147483648
    - **De valójában nem definiált**

Nem definiált viselkedés (folyt.)

Undefined behavior

```
bool overflow(int i) {
  return i+1 < i;
}

int main() {
  int i = INT_MAX;
  if (overflow(i))
    printf("overflow\n");
  else
    printf("%d\n", i+1);
}
```

Mi lesz az eredmény?

Nem definiált viselkedés (folyt.)

Undefined behavior

```
bool overflow(int i) {
  return i+1 < i;
}

int main() {
  int i = INT_MAX;
  if (overflow(i))
    printf("overflow\n");
  else
    printf("%d\n", i+1);
}
```

Fordító szempontjából

- Ha `i` $<$ `INT_MAX` → hamis
- Ha `i` $==$ `INT_MAX`
  - Előjeles túlcsordulás lesz
  - Az eredmény bármi lehet

```
bool overflow(int i) {
  return false;
}
```

Mi lesz az eredmény?

- **Az overflow minden esetben hamissal tér vissza**
  - -2147483648

C++ szabvány

- A C++23 szabvány
  - 2.134 oldal hosszú
  - *Undefined behavior* - 116 találat
  - *Unspecified* - 812 találat
- Nem definiált viselkedés
  - Nem inicializált változó
  - Hiányzó visszatérési érték
  - Nullpointer dereferencia
  - Tömb túlindexelés
  - „Rossz” memória használat

```
class Macska {
public:
  void almodik() {
    cout << "A macska almodik" << endl;
  }
};

int main() {
  Macska *macska = nullptr;
  macska->almodik();
}
```

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

int main() {
  vector<int> v {1, 3, 8};
  v[3] = 12;
  cout << v[0] << " " << v[3] << endl;
}
```

Tartalom

I/O műveletek

- A C++ Standard Template Library (STL) része
  - Folyamok (`stream`-ek) segítségével valósítja meg
- Szabványos bemenet/kimenet (`iostream` header)
  - `cin` szabványos bemenet folyam
  - `cout` szabványos kimenet folyam
  - `cerr` szabványos hiba (kimenet) folyam
  - `clog` szabványos logging (kimenet) folyam
- Fájl írása, olvasása (`fstream` header)
  - `ifstream` - stream fájlok olvasására
  - `ofstream` - stream fájlok írására
  - `fstream` - stream fájlok olvasására és írására

C++ szabványos bemenet/kimenet

```cpp
#include <iostream>


int main() {
  int i, j;
  std::cin >> i >> j;
  std::cout << i << "*" << j
    << " = " << i*j << std::endl;
}
```

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

int main() {
  int i, j;
  cin >> i >> j;
  cout << i << "*" << j
       << " = " << i*j << endl;
}
```

- `iostream` header fájlt kell include-olni
- Az `std` névtérben találhatóak az I/O folyamok
  - Scope-olt névvel kell rájuk hivatkozni
    - `std::` (a `::` a scope operátor)
- A `>>` és `<<` operátorok segítségével olvasunk és írunk
  - Az operátor az adott olvasás/írás után a `stream`-mel tér vissza, azaz újból alkalmazhatjuk rá az operátort
    - `int i = a * b * c * d; // a, b, c, d egesz`
- Az `endl` a sorvége

Különböző kimeneti folyamok

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

int main() {
  cout << "Hello ";
  cerr << "Vilag!" << endl;
}
```

- „Szokásos futtatás” után a kimenet: `Hello Vilag!`
  - A `cout` és `cerr` folyamok kimenetei „összemosódnak”
    - De attól külön folyamok
- Felhasználás külön-külön
  - `./hw 1>out.txt 2>error.txt`

Tartalom

C++ típusok

Csoport	Típus neve	Min. méret (bit)
Karakter	`char`	8
	`char16_t`	16
	`char32_t`	32
Egész típusok (előjeles)	`signed char`	8
	signed `short` int	16
	signed `int`	16
	signed `long` int	32
	signed `long long` int	64
Egész típusok (előjel nélküli)	`unsigned char`	8
	`unsigned short` int	16
	`unsigned int`	16
	`unsigned long` int	32
	`unsigned long long` int	64
Lebegőpontos	`float`	ált. 32
	`double`	ált. 64
	`long double`	ált. 80
Logikai típus	`bool`	implementáció-függő
Void típus	`void`	
Nullpointer típus	`std::nullptr_t`	

$1 = \text{sizeof(char)} \leq \text{sizeof(short)} \leq \text{sizeof(int)} \leq \text{sizeof(long)} \leq \text{sizeof(long long)}$

std::string

- A C++ Standard Template Library (stl) része
  - Az `std` névtérben található

```
#include <string>
#include <iostream>

using namespace std;

int main() {
  string s1 = "Hello";
  string s2("Vilag");
  string s3;

  s3 = s1 + " " + s2 + "!";
  cout << s3 << endl;  // Hello Vilag!
}
```

C++ string

Konstruktor	`string();`
	`string(const string& str);`
	`string(const char* s);`

Méret (hossz)	`size_t size();` `size_t length();`
Üres-e?	`bool empty();`
Törlés	`void clear();`
Rész string	`string substr(size_t p = 0, size_t l = npos);` `p` a kezdő pozíció, `l` a hossz
Első karakter	`char& front();`
Utolsó karakter	`char& back();`
i-edik karakter	`char& operator[] (size_t pos);` `char& at(size_t pos);`

Keresés	`size_t find(const string& str, size_t pos = 0);`
	`size_t find_first_of(char c, size_t pos = 0);`
	`find_last_of`, `find_first_not_of`, `find_last_not_of`

„Vége”	`npos` attribútum
`char*` repr.	`const char* c_str();`

- Bővebben: `https://en.cppreference.com/w/cpp/string/basic_string`

C++ string

```
#include <iostream>       // bemenet
#include <string>         //   programozas
                          // kimenet
using namespace std;      //   programozas hossza: 11
                          //   programozas elso karaktere: p
int main() {              //   'a' elso elofordulasa: 5
  string s;               //   [2-4]: rog
  cin >> s;               //   torles utan: ''
  cout << s << " hossza: " << s.length() << endl;
  if (!s.empty())
    cout << s << " elso karaktere: " << s[0] << endl;
  size_t pos = s.find_first_of('a');
  if (pos != string::npos)
    cout << "'a' elso elofordulasa: " << pos << endl;
  if (4 < s.length())
    cout << "[2-4]: " << s.substr(1, 3) << endl;
  s.clear();
  cout << "torles utan: '" << s << "'" << endl;
}
```

Szám konverziója string-gé

- A `string` és a szám típusok között nincs automatikus konverzió

```
string s = 5; // HIBA!!!
```

- `string to_string(int value);`
  - Nem a string osztály metódusa (de az `std` névtérben van)
  - Minden szám típusra meg van valósítva
    - `string to_string(long long value);`
      `string to_string(double);`
      ...

- Példa

```
int main() {
  int i = 5;
  string s1 = to_string(i);
  string s2 = to_string(3.14);
  cout << s1 << " " << s2 << endl;
}
// 5 3.140000
```

string konverziója számmá

- A string típus nem konvertálódik automatikusan egésszé

```
string str("4");
int i = str; // HIBA!
```

- `int stoi(const string& str, size_t* pos = 0, int base = 10);`
  - str: a string, amit számmá kell konvertálni
  - pos: hány karakter lett feldolgozva
  - base: a számrendszer alapja (2, 3, 4, ..., 36)
    - A 0 azt jelenti, hogy automatikusan kell felismernie
- Példa

```
int main() {
  string s1("12345");
  int i = stoi(s1);
  cout << i << endl;
}
// 12345
```

string konverziója számmá (folyt.)

- Példa

```
int main() {
  string s1("+FF alma");
  unsigned int chars = 0;

  int i = stoi(s1, &chars, 16);
  cout << i << " " << chars << endl;
}
```

- Eredmény: 255 3
  - 16-os számrendszerben az FF az 255 10-es számrendszerben
  - 3 karakter lett feldolgozva (a + jel is)
- További konverziók

`int`: `stoi`	`unsigned long`: `stoul`
`long`: `stol`	`unsigned long long`: `stoull`
`float`: `stof`	`long long`: `stoll`
`double`: `stod`	`long double`: `stold`

Hibakezelés

- Kétféle hiba léphet fel
  - Nem lehet konvertálni
    - `std::invalid_argument` kivétel dobódik

```
try {
  string s1("alma");
  cout << stoi(s1) << endl;
} catch (const invalid_argument& e_inv) {
  cout << "invalid_argument exception" << endl;
}
```

  - Nem fér bele az adott tartományba
    - `std::out_of_range` kivétel dobódik

```
try {
  string s1("99999999999999999999999999999");
  cout << stoi(s1) << endl;
} catch (const out_of_range& e_out) {
  cout << "out_of_range exception" << endl;
}
```

Tartalom

„Mint Javaban”

- Tabulálás nem számít, **de törekedjünk az olvasható kódra**
  - Rossz tabulálás
    - A fordítóprogram el tudja olvasni
    - A fejlesztőnek lesz nehezebb megérteni
- Vezérlési szerkezetek „megegyeznek”
  - Elágazások
    - `if else`
    - `switch case`, van `default` ág is
  - Ciklusok
    - `for`
    - `while`, `do while`
- Néhol bővebb

```
int main() {
  string nev;
  cin >> nev;
  if (size_t pos = nev.find('a'); pos != string::npos)
    cout << "Az 'a' elso eloforudlasa: " << pos << endl;
  else
    cout << "Nem szerepel benne 'a' karakter." << endl;
}
```

Tartalom

Tartalom

Objektum-orientált programozás

- Egységbezárás
  - Az összetartozó adatokat és a rajtuk végezhető műveleteket egy egységként kezeljük
- Öröklődés
  - Az osztályok öröklődéssel újrafelhasználhatóak
  - Új adattagokkal és operációkkal kibővíthetőek
  - Meglévő operációkat felül lehet definiálni
- Polimorfizmus
  - Futás közben dől el, hogy pontosan melyik operáció hajtódik végre

Objektum

- Az objektum egy entitás ábrázolása
  - Kutya, macska, ember, kurzus, tárgy, ...

Minden objektumnak van

- Állapota
  - Egy a lehetséges létezései közül
  - Időben változó
  - *Attribútumok* határozzák meg
    - Példa: a macskának van színe és születési dátuma
- Viselkedése
  - Annak módja, hogyan reagál egy objektum más objektumok kéréseire
  - Mindent definiál, amit az objektum „csinálhat”
  - *Operációk* határozzák meg
    - Egy `Macska` típusú objektumnak lehet `jatszik` vagy `almodik` operációja
- Identitása
  - Minden objektum egyedi, akkor is, ha az állapotuk azonos
    - Lehet két azonos nevű macska, de azok attól különbözők
    - A típusuk és az attribútumaik azonosak

Osztály fogalma

- Leírás objektumok csoportjához, amelyeknek közösek az
  - Attribútumaik, operációik
  - Más objektumokkal való kapcsolataik és
  - Szemantikus viselkedésük
- Egy minta objektumok létrehozásához (példányosításához)
  - Minden objektum pontosan egy osztály példánya
  - Az osztály az objektum típusa

C++ osztály

- Meg kell adni az osztály
  - nevét
  - attribútumait (és azok típusait)
    - `int szulEv;`
      `string szin, nev;`
  - tagfüggvényit
    - deklaráció: `void almodik();`
    - definíció: `void jatszik() { /* ... */ }`

```
class Macska {
  string szin;          // adattag (attributum)
  int szulEv;           // adattag (attributum)

  void jatszik() {      // metodus (tagfuggveny)
    cout << "Bogrevel jatszik" << endl;
  }

  void almodik();       // metodus deklaracio
}; // pontosvesszo!
```

Objektum létrehozása

- Ha definiáltunk egy osztályt, akkor azt példányosíthatjuk
  - Megfelelő konstruktor meghívódik
  - Létrejön egy új objektum
    - Típusa: az adott osztály
- Minden objektum példány egyedi
  - `m1` és `m2` különbözők
- Az adott objektumnak
  - Meg lehet hívni a metódusait
    - `cirmi.jatszik();`
  - Hozzáférünk az attribútumaihoz
    - `m.szin = "fekete";`
    - `cout << cirmi.szin;`

```
class Macska {
public:
  string szin, nev;
  int szulEv;

  Macska() {
    szulEv = 2024;
  }

  Macska(string s, string n) {
    szin = s;
    nev = n;
    szulEv = 2024;
  }

  void jatszik() {
    cout << "..." << endl;
  }

  void almodik();
};

int main() {
  Macska m1, m2;
  macska1.szin = "fekete";

  Macska cirmi("cirmos", "Cirmi");
  cirmi.jatszik();
  cout << cirmi.szin << endl;
}
```

Láthatóság

- Korlátozhatjuk, hogy az osztályból mihez férjenek hozzá
  - Az osztály tagjainak meg lehet adni a láthatóságát
- Láthatóságok
  - `public`: mindenki számára korlátozás nélkül elérhető
  - `protected`: csak az adott osztályból és a leszármazott osztályokból érhető el
  - `private`: csak az adott osztályból érhető el
- `class` esetében az alapértelmezett láthatóság `private`
- „Szekcióra” érvényes
  - Addig, amíg újat nem definiálunk

```
class Macska {
  string szin, nev; // private
private:
  int szulEv;

protected:
  void setNev(string n) {
    nev = n;
  }

public:
  Macska() {
    szulEv = 2024;
  }

  Macska(string s, string n) {
    szin = s;
    nev = n;
    szulEv = 2024;
  }

  string getNev() {
    return nev;
  }

public: // felesleges, de nem hiba
  void jatszik() { /* ... */ }

  void almodik();
};
```

Ki olthatja be a macskát?

- Szeretnénk megvalósítani, hogy a macskát be lehessen oltani
  - Legyen egy `beolt()` metódusa a macskának
- Szeretnénk, ha csak az orvos olthatná be
  - Milyen legyen a láthatósága a metódusnak?

`public`

`protected`

`private`

- A láthatóság azt mondja meg, hogy mi érhető el az osztályból
  - Nekünk az kellene, hogy ki vagy mi érheti el

friend mechanizmus

- Lehetőség van hozzáférést biztosítani az osztályhoz másik osztályoknak, függvényeknek vagy metódusoknak
  - A `friend` kulcsszó használatával adhatjuk meg
    - Az adott osztályon belül kell megadni
  - **Mindenhez hozzáfér**
    - Nem tudjuk korlátozni, hogy csak „bizonyos” tagokhoz férjen hozzá

```
class Macska {
  /* ... */
  friend void beolt(Macska& k) { /* ... */ }
};
```

- Annak ellenére, hogy az adott osztályon belül adjuk meg a `friend`-eket, azok **nem az osztály részei**

```
int main() {
  Macska macska;
  macska.beolt(macska); // HIBA!!!
}
```

```
int main() {
  Macska macska;
  beolt(macska);
}
```

friend függvény

Példa

```
class Macska {
  int egeszseg = 100;
  void beolt() { egeszseg++; }

  friend void orvosBeolt(Macska& m);
  friend void allatorvosBeolt(Macska& m) { m.egeszseg++; }
};

void orvosBeolt(Macska& m) { m.beolt(); }
```

```
int main() {
  Macska macska;

  // HIBA, az egeszseg private
  macska.egeszseg++;

  // HIBA, nincs ilyen metodus
  macska.orvosBeolt();
  macska.allatorvosBeolt(macska);
}
```

```
int main() {
  Macska macska;

  orvosBeolt(macska);
  allatorvosBeolt(macska);
}
```

friend osztály

Példa

```
class Macska {
  int egeszseg = 100;
  void beolt() { egeszseg++; }
  friend class Allatorvos;
};
```

```
class Gazda {
public:
  void beolt(Macska& m) {
    m.beolt();      // HIBA
    m.egeszseg++;   // HIBA
  }
};
```

```
class Allatorvos {
public:
  void beolt(Macska& m) {
    m.beolt();
    //m.egeszseg++; is lehetne
  }
};
```

```
int main() {
  Macska m;
  Gazda gazda;
  gazda.beolt(m);
}
```

```
int main() {
  Macska m;
  Allatorvos orvos;
  orvos.beolt(m);
}
```

this

- A `this` segítségével hivatkozhatunk az adott objektumra
  - **Pointer**, ami az objektumra mutat
    - Automatikusan beállítódik
- Hol és hogyan használjuk?
  - Névütközések „feloldása”

```
class Macska {
  unsigned ev;
  string szin;
public:
  Macska(string szin, int ev) {
    this->ev = ev;
    this->szin = szin;
  }
};
```

  - Elkerülhető
    - Különböző nevek használatával
  - Objektum visszaadása

```
class Macska {
  // ...
public:
  Macska& eszik(Etel etel) {
    // ...
    return *this;
  }

  Macska& jatszik() {
    // ...
    return *this;
  }
};

int main() {
  Macska macska;
  macska.jatszik().eszik(kaja);
}
```

Operator overloading

- Ugyanaz az művelet, de más a paraméter
  - Miért adjunk másik nevet, majd a paraméter eldönti
  - Konstruktor esetében nem is lehet másik név

```
class Labda { };

class Macska {
public:
  void jatszik() {
    cout << "A macska ugral" << endl;
  }

  void jatszik(Labda labda) {
    cout << "A macska a labdaval jatszik" << endl;
  }
};

int main() {
  Macska cirmi;
  cirmi.jatszik();

  Labda labda;
  cirmi.jatszik(labda);
}
```

Tartalom

Osztályok példányosítása

Inicializálás

- Az osztályok példányosításával objektumokat hozhatunk létre
  - Az objektumok van állapota
    - Az adattagjai határozzák meg
- Az adattagokat inicializálni kell az objektum létrejöttekor
  - Lehetne két lépésben is, hogy előbb létrehozzuk, majd inicializáljuk
    - A két lépés felesleges
  - Vannak esetek, amikor ez nem is lehetséges
    - `const` adattagokat létrehozás után nem lehet módosítani
    - Referenciát kötelező inicializálni

Konstruktor

- Az osztálynak definiálhatunk konstruktort, amely az osztály példányosításakor automatikusan lefut
- A konstruktor egy speciális tagfüggvény
  - A neve meg kell egyezzen az osztály nevével
  - Nem lehet visszatérési típusa
  - Nem hívható közvetlenül (kivéve delegating konstruktor)
    - Automatikusan hívódik
  - Nem lehet `const` vagy `static`
- Viszont általános olyan tekintetben, hogy
  - Megadhatjuk a láthatóságát
  - Lehetnek paraméterei
  - Lehet több konstruktor is
    - Operator overloading
- A paraméter nélküli konstruktort default konstruktornak hívják
  - Ha nem adunk meg konstruktort, akkor a fordító generál egy default-ot
    - Ha van „nem triviálisan inicializálható” adattag, akkor nem generál

Konstruktor példa

```
class Macska {
  string szin;
  int szulEv;

public:
  Macska() {
    szin = "ismeretlen";
    szulEv = 2024;
  }

  Macska(string sz) {
    szin = sz;
    szulEv = 2024;
  }

  Macska(int ev) {
    szin = "ismeretlen";
    szulEv = ev;
  }

  Macska(string sz, int ev) {
    szin = sz;
    szulEv = ev;
  }
};
```

```
int main() {
  Macska m1;
  Macska m2("cirmos");
  Macska m3(2018);
  Macska m4("fekete", 2017);
}
```

- Lefordul, de ...
  - ez függvény deklaráció

```
int main() {
  Macska macska();
}
```

Konstruktor (folyt.)

- Csak azokat a konstruktorokat hívhatjuk, amelyek definiálva vannak
  - Paraméterlista alapján dől el, hogy melyik hívódik
    - Ahogy a metódusok (függvények) esetében is

```
class Macska {
  string szin;
  int szulEv;
public:
  Macska(string sz, int ev) {
    szin = sz;
    szulEv = ev;
  }
};

int main() {
  Macska m1;              // HIBA! nincs megfelelo konstruktor
  Macska m2("cirmos"); // HIBA! nincs megfelelo konstruktor
  Macska m3(2018);        // HIBA! nincs megfelelo konstruktor
  Macska m4("fekete", 2017);

}
```

Osztály példányosítása

Objektum létrejötte

- Ha létrehozunk egy objektumot, akkor
  1. Lefoglalódik a szükséges memória
  2. **Inicializálódnak az adattagok**
     - Vagy inicializálatlanok maradnak
  3. Lefut a konstruktor törzse

```
class Macska {
  string szin;
  int szulEv;
public:
  Macska(string sz, int ev) {
    szin = sz;
    szulEv = ev;
  }
};
```

```
class Datum {
  unsigned ev, ho, nap;
public:
  Datum(unsigned e) {
    ev = e;
    ho = nap = 1;
  }
};

class Macska {
  Datum szulEv;
  string szin;
public:
  Macska(string sz, int ev) {
    szulEv = Datum(ev);
    szin = sz;
  }
};
```

Konstruktor inicializáló lista

- Az adattagokat a **konstruktor inicializáló listában** kell (lehet) inicializálni
  - A paraméter lista után kettőspont (:)
  - Majd az adattagok felsorolva és zárójelben az értékük
  - A felsorolást vesszővel kell elválasztani

```
class Datum {
  unsigned ev, ho, nap;
public:
  Datum(unsigned e) {
    ev = e;
    ho = nap = 1;
  }
};

class Macska {
  Datum szulEv;
  string szin;
public:
  Macska(int ev, string sz) :
    szulEv(ev), szin(sz)
  {
  }
};
```

```
class Datum {
  unsigned ev, ho, nap;
public:
  Datum(unsigned e) {
    ev = e;
    ho = nap = 1;
  }
};

class Macska {
  Datum szulEv;
  string szin;
public:
  Macska(int szulEv, string szin) :
    szulEv(szulEv), szin(szin)
  {
  }
};
```

Konstruktor inicializáló lista sorrendje

- **Az adattagok abban a sorrendben kapnak értéket, ahogy az osztályban definiálva vannak**
  - Azaz nem abban a sorrendben hajtódik végre az inicializáló lista, ahogy a listában fel vannak sorolva az adattagok

```
class Datum {
  unsigned ev, honap, nap;
public:
  Datum() : nap(1), honap(nap), ev(honap) {
    // ev = honap;  // HIBA! honap definialatlan
    // honap = nap; // HIBA! nap definialatlan
    // nap = 1;
  }
};
```

  - A fordító warning üzenetet ad

Adattagok inicializálása

Default member initialization

- Az adattagoknak lehet kezdőértéket adni
  - Ha a konstr. inici. listában nem kap értéket, akkor ez lesz az értéke

```
class Datum {
  unsigned ev = 2023, honap = 1, nap = 1;

public:
  Datum() { }

  Datum(unsigned ev) : ev(ev) { }

  Datum(unsigned ev, unsigned honap, unsigned nap) :
    ev(ev), honap(honap), nap(nap) { }

  void kiir() { cout << ev << " " << ho << " " nap << endl; }
};
int main() {
  Datum d0;
  d0.kiir();              // "2023 1 1"

  Datum d1(2000);
  d1.kiir();              // "2000 1 1"

  Datum d2(2023, 9, 19);
  d2.kiir();              // "2023 9 19"
}
```

Delegating konstruktor

- Az egyik konstruktorból meg lehet hívni a másikat
  - A konstruktor inicializáló listában kell megadni, hogy melyiket hívjuk
    - Ilyenkor nem inicializálhatjuk az adattagokat
  - Előnye, hogy nem kell a kódot duplikálni
    - Kevesebb a hibalehetőség

```
class Datum {
  unsigned ev, honap, nap;
public:
  Datum() : Datum(2023, 9, 19) {
  }

  Datum(unsigned ev, unsigned honap, unsigned nap) :
    ev(ev), honap(honap), nap(nap) {
    /* hosszu, bonyolult algoritmus */
  }
};
```

Nem publikus konstruktor

- A konstruktor láthatóságát is változtathatjuk
  - Lehet például `private` is
    - Ilyenkor az osztályon kívülről nem lehet példányosítani
- Mire jó, ha kívülről nem lehet példányosítani?
  - Akkor hogyan használjam?

```
class Singleton {
  Singleton() {}
public:
  static Singleton& getInstance() {
    static Singleton instance;
    return instance;
  }
};
```

```
int main() {
  // HIBA!!!
  Singleton s2;
}
```

```
int main() {
  Singleton s1 = Singleton::getInstance();
  Singleton s2 = Singleton::getInstance();
  // s1 es s2 ugyanaz az objektum
}
```

Példányosítás megakadályozása

- Mi van akkor, ha meg akarom tiltani a példányosítást
  - A nem publikus konstruktor nem jó megoldás

```
class NemPeldanyosithato {
  NemPeldanyosithato() { }
public:
  void foo() {
    NemPeldanyosithato e;  // megis peldanyositottam
  }
};
```

  - delete konstruktor

```
class NemPeldanyosithato {
  NemPeldanyosithato() = delete;
public:
  void foo() {
    NemPeldanyosithato e;  // HIBA!
  }
};
```

Default konstruktor hiánya

- Nincs default konstruktor

```
class Macska {
  string szin;
public:
  Macska(const string& sz) : szin(sz) { }
};
```

  - Nem példányosíthatjuk úgy, hogy a default konstruktor hívódik

```
Macska macska;
```

  - Van, amikor automatikusan hívódna
    - A fordítóprogram generálna default konstruktort a `Tulajdonosnak`
    - Ami hívná a `Macska` default konstruktorát, de az hiányzik

```
class Tulajdonos {
  Macska macska;
};
```

  - Tömböt sem tudunk létrehozni
    - 5 `Macska` példány jönne létre, de a default konstruktorok hívódnának

```
Macska macskak[5];
vector<Macska> macskak(5);
```

Default konstruktor hiánya

Hogyan példányosítsunk?

- Ha van értelme, akkor megvalósíthatjuk a default konstruktort
  - Nem minden esetben tehetjük meg
    - Nem saját osztály, csak használjuk
    - Nincs értelme, mert „valamit” inicializálni kell

```
class Macska {
  string nev;
public:
  Macska(string n) : nev(n) { }
};
```

- Használjuk a megfelelő konstruktort
  - `Macska macska("cirmi");`
  - A konstruktor inicializáló listában kell meghívni a megfelelő konstruktort

```
class MacskaTulajdonos {
  Macska macska;
public:
  MacskaTulajdonos(string nev) : macska(nev) { }
};
```

- Tömbök, tárolók inicializálásáról később

Tartalom

Hány macskát látunk?

- Feladat: valósítsuk meg, hogy a gazdi megsimogathassa a macskát
  - A macska ettől boldogabb lesz

```
class Macska {
  unsigned jokedv = 5;
public:
  unsigned getJokedv() {
    return jokedv;
  }

  void simogat() {
    if (jokedv < 100)
      jokedv++;
  }
};

class Gazda {
public:
  void simogat(Macska m) {
    m.simogat();
  }
};

int main() {
  Gazda gazdi;
  Macska cirmi;
  cout << cirmi.getJokedv() << endl;
  gazdi.simogat(cirmi);
  cout << cirmi.getJokedv() << endl;
}
```

Hány macskát látunk?

- Feladat: valósítsuk meg, hogy a gazdi megsimogathassa a macskát
  - A macska ettől boldogabb lesz
- Két macska van „elrejtve” a kódban
  - A hívásnál lemásolódik
    - A lemásolt macskát eteti meg (módosítja) a gazdi
  - Az eredeti nem változik
- A futás eredménye

  5

  5
- Cél
  - Nem lemásolni a macskát
  - Az eredetit átadni
    - A gazdi azt eteti meg

```
class Macska {
  unsigned jokedv = 5;
public:
  unsigned getJokedv() {
    return jokedv;
  }

  void simogat() {
    if (jokedv < 100)
      jokedv++;
  }
};

class Gazda {
public:
  void simogat(Macska m) {
    m.simogat();
  }
};

int main() {
  Gazda gazdi;
  Macska cirmi;
  cout << cirmi.getJokedv() << endl;
  gazdi.simogat(cirmi);
  cout << cirmi.getJokedv() << endl;
}
```

Érték vagy referencia szerinti átadás

- Érték szerint

```
class Gazda {
public:
  void megetet(Macska m) {
    m.megetet();
  }
};

int main() {
  Gazda gazdi;
  Macska cirmi;
  gazdi.megetet(cirmi);
}
```

- Az objektum lemásolódik
  - Létrejön egy új `Macska` (`m`)
  - Az állapota megegyezik az eredetivel
- A másolatot változtatjuk meg
  - Az eredeti macska (cirmi) nem változik

- Referencia szerint

```
class Gazda {
public:
  void megetet(Macska& m) {
    m.megetet();
  }
};

int main() {
  Gazda gazdi;
  Macska cirmi;
  gazdi.megetet(cirmi);
}
```

- A referencia egy „alias”
  - Egy adott objektumnak „két neve van”
- cirmi: main-ben egy Macska típusú objektum
  - m: „hivatkozás” cirmi-re

Referencia paraméter

- Ha módosítani szeretnénk az „eredeti” objektumot, akkor referencia

```
class Gazda {
  void megetet(Macska& m) {
    m.megetet();
  }
};

int main() {
  Gazda gazdi;
  Macska cirmi;
  gazdi.megetet(cirmi);
}
```

- Az objektum nem másolódik le
  - Performancia szempontjából optimálisabb
    - Futásidő, memória
- Megváltoztathatjuk az „eredeti” objektumot
  - Nem kell lemásolni és a módosítottat visszamásolni

„Érték” szerinti paraméterátadás

- „Érték” szerint
  - Csak felhasználjuk az értékét
  - De az „eredeti” objektumot nem akarjuk megváltoztatni
- Kicsi objektum esetén „érték” szerint
  - Nem probléma a másolás

```
class Macska {
  Macska(int szulEv);
};
```

- Ha „nagy” az objektum, akkor konstans referencia
  - Referencia miatt nem másolja le az objektumot
  - `const` miatt nem módosíthatja

```
class Allatorvos {
public:
  void megvizsgal(const Macska& macska);
};
```

- Ha a paraméterben kapott objektumot módosítani akarom, de az eredetit nem
  - Akkor érték szerint

Referencia

Visszatérési érték, adattag, (lokális) változó

- Visszatérési érték

```
class Macska {
  string nev;
public:
  string& getNev() {
    return nev;
  }
};
```

```
int main() {
  Macska macska;
  macska.getNev() = "Cirmi";
}
```

- Adattag

```
class MacskaTulajdonos {
  Macska &macska;
public:
  MacskaTulajdonos(Macska &m) : macska(m) {}
};
```

- (Lokális) változó

```
string &nev = macska.getNev();
cout << nev << endl;
```

Referencia - problémák

- Létrehozáskor értéket kell adni a referenciának
  - Paraméternél ez nem probléma
  - Ref. adattagot a konst. inic. listában inicializálni kell
    - Fordító nem is tud default konstruktort generálni

```
class MacskaTulajdonos {
  Macska &macska;
public:
  MacskaTulajdonos(Macska &m) /* : macska(m) kellene */ {
    macska = m; // mar keso
  }
};
```

- Nem lehet megváltoztatni, hogy minek a referenciája

```
int i = 5, j = 6;
int &k = i;
k = j; // k (es i) felveszi j erteket
```

- Referencia paraméter, de az argumentum nem objektum

```
class Macska {
  int szulEv;
public:
  void setSzulEv(int& ev) {
    szulEv = ev;
  }
};
```

```
int main() {
  Macska macska;
  int i = 2024;
  macska.setSzulEv(i); // OK
  macska.setSzulEv(2024); // HIBA
}
```

Tartalom

Konstans

- `const`
  - Jelezhetem, hogy valamit nem akarok megváltoztatni
- Miért használjam?
  - Ha mégis megváltozna, akkor szól a fordító
    - Fordításkor kiderül a **hiba**
  - Információ a fejlesztőnek
    - Tudom, hogy nem akartam megváltoztatni
  - Segítség a „felhasználónak”
    - Ha használják a kódomat, tudják, hogy mikor nem változik „valami”
    - Például egy paraméter
  - `const` objektumra is hívható legyen a metódus

Hol használjuk?

Változók, objektumok

- Lokális változó

```
int main() {
  const unsigned MACSKA_DB = 10;
  for (unsigned i = 0; i < MACSKA_DB; i++) {
    // beolvas, eltarol
  }
  // ...
}
```

- Osztály adattag

```
class MacskaMenhely {
  const unsigned KAPACITAS = 30;
  vector<Macska> macskak;

public:
  void add(Macska m) {
    if (KAPACITAS == macskak.size()) {
      // hibakezeles
    }
    // macska befogadasa
  }
};
```

```
class MacskaMenhely {
  const unsigned KAPACITAS = 30;
  vector<Macska> macskak;

public:
  void add(Macska m) {
    if (KAPACITAS = macskak.size()) {
      // hibakezeles
    }
    // macska befogadasa
  }
};
```

Hol használjuk?

Paraméter, visszatérési típus

- Konstans referencia használata
  - Nem másoljuk, de nem is lehet megváltoztatni
- Visszatérés típusa

```
class Macska {
  string szin;
public:
  /* ... */
  const string& getSzin() {
    return szin;
  }
};
```

- Paraméter

```
unsigned megszamol(const vector<Macska>& m, const string& szin) {
  unsigned db = 0;
  for (size_t i = 0; i < m.size(); i++)
    if (m[i].getSzin() == szin)
      ++db;
  return db;
}
```

Konstans metódus

- Osztályból sok objektum keletkezik
  - Különböző memóriaterületen tárolódnak
- Egy metódus csak egy példányban van a memóriában
  - Honnan tudja, hogy melyik objektumhoz tartozik?

Konstans metódus

- Osztályból sok objektum keletkezik
  - Különböző memóriaterületen tárolódnak
- Egy metódus csak egy példányban van a memóriában
  - Honnan tudja, hogy melyik objektumhoz tartozik?
- A `this` automatikusan beállítódik az objektumra

```
unsigned getSzulEv() {
  return szulEv;
}
```

```
unsigned getSzulEv() {
  return this->szulEv;
}
```

- Ezt a metódus is használja
  - Megkapja, mint „rejtett” paraméter
- A `this` is lehet konstans
  - A metódus paraméterlista után kell megadni

```
void setSzulEv(unsigned s) const {
  szulEv = s;
  // this->szulEv = s;
}
```

```
unsigned getSzulEv() const {
  return szulEv;
  //return this->szulEv;
}
```

Konstans metódus

Mit lehet és mit nem?

- Nem változtathatja meg az objektumot

```
void setSzulEv(unsigned s) const {
  this->szulEv = s;
}
```

```
unsigned getSzulEv() const {
  return szulEv;
}
```

- Nem hívhatja meg a nem `const` metódusát az **objektumnak**

```
class Macska {
  string nev;
public:
  string getNev() {
    return nev;
  }
  void kiir() const {
    cout << "Macska: " << getNev();
    // this->getNev()
  }
};
```

```
class Macska {
  string nev;
public:
  string getNev() const {
    return nev;
  }
  void kiir() const {
    cout << "Macska: " << getNev();
    // this->getNev()
  }
};
```

- Nem adhat vissza nem `const` referenciát az adattagra

```
string& getNev() const {
  return this->nev;
}
```

```
const string& getNev() const {
  return this->nev;
}
```

Ki kit hívhat?

- Nem `const` bármit hívhat
  - `const`-ot is

```
// nem const objektum
Macska m;
// const metodus hivasa
cout << m.getNev();
// nem const metodus hivasa
m.setNev("Sicc");
```

- `const` csak `const`-ot hívhat

```
void kiir(const Macska& c) {
  // const obj. const hivas
  cout << "cica: " << c.getNev();
}
```

```
void keresztel(const Macska& c) {
  // const obj. nem const hivas
  c.setNev("Mirr-murr");
}
```

```
class Macska {
  string nev;
public:
  const string& getNev() const {
    return nev;
  }

  void setNev(const string& n) {
    nev = n;
  }
};
```

Mikor melyik hívódik?

- `this`, mint „rejtett” paraméter
  - Erre is alkalmazható az operáció kiterjesztés
    - Lehet egyszerre `const` és nem `const` metódus is

```
class Macska {
  string nev = "Cirmi";
public:

  const string& getNev() const {
    cout << "const, ";
    return nev;
  }

  string& getNev() {
    cout << "nem const, ";
    return nev;
  }
};
```

Nem `const` a nem `const`-ot h.

```
int main() {
  Macska m;
  cout << m.getNev() << endl;
  // nem const, Cirmi
  m.getNev() = "Sicc";
  cout << m.getNev() << endl;
  // nem const, Sicc
}
```

`const` csak a `const`-ot hívhatja

```
int main() {
  const Macska c;
  cout << c.getNev() << endl;
  // const, Cirmi
}
```

```
  m.getNev() = "Sicc"; // HIBA
```

Logikailag konstans

- A konstans szónak kétféle értelmezése is lehet
- Semmi sem változhat, azaz bájtra pontosan ugyanaz marad az objektum
  - Programozás szempontjából ezt jelenti a `const`
    - Ezt láttuk a korábbi példákban
- Logikailag nem változik, azaz az objektum „érdemben” nem változik
  - A felhasználó számára „lényeges adatok” nem változnak
  - De a belső működéséhez szükséges adattagok változhatnak
- Példa: online bankrendszer
  - Egyenleg lekérdezése a felhasználó számára „logikailag konstans” művelet
    - Csak megnézi az egyenleget, de ne változzon meg
  - A bank azért feljegyezheti az utolsó belépés vagy lekérdezés időpontját
    - Olyan adat változik, ami a felhasználó számára „nem fontos”
- Programozói szemszögből
  - Olyan `const` metódust kell írni, amely mégis módosíthatja az osztály adattagját
  - Ezt a `mutable` kulcsszó segítségével tehetjük meg, amit a deklarációnál kell megadni

Példa

HIBÁS!!!

- `const` metódus változtatja az objektumot

```
class Datum {
public:
  static Datum getAktualisDatum() { /* ... */ }
};

class Bankszamla {
  int egyenleg = 0;
  Datum utolsoMuvelet = Datum::getAktualisDatum();

public:

  int getEgyenleg() const {
    utolsoMuvelet = Datum::getAktualisDatum(); // HIBA
    return egyenleg;
  }
};
```

mutable

- `mutable` adattag módosítható `const` objektum esetén is

```
class Datum {
public:
  static Datum getAktualisDatum() { /* ... */ }
};

class Bankszamla {
  int egyenleg = 0;
  mutable Datum utolsoMuvelet = Datum::getAktualisDatum();

public:

  int getEgyenleg() const {
    utolsoMuvelet = Datum::getAktualisDatum();
    return egyenleg;
  }
};
```

Tartalom

C++ tárolók, algoritmusok

- Típusfüggetlenség
  - Példányosításkor elég megmondani a típust
  - Algoritmusok működése független a tárolótól
    - Iterátorok használata
- Hatékonyság
  - A cél függvényében választhatunk tárolót
    - Nem is mindig lehet akármelyiket választani
- Implementáció
  - Nem nekünk kell megírni
  - Hatékony, tesztelt, biztonságos, ...
- Széles körben elterjedt
  - Sokan ismerik és használják

Macska osztály

```
class Macska {
  string szin = "fekete";
public:
  Macska(const string& s) : szin(s) {
  }

  Macska() = default;

  const string& getSzin() const {
    return szin;
  }

  void setSzin(const string& s) {
    szin = s;
  }

  void kiir() const {
    cout << szin << " ";
  }
};
```

Tartalom

vector

- `https://en.cppreference.com/w/cpp/container/vector`
- `vector<T> v;`
  - `T` a típus
- `T` elemekből álló tömböt hoz létre
  - Az elemeket „egymás mellett” tárolja
    - Mintha tömb lenne
  - Az elemszámra nincs felső korlát
    - Csak a memória mérete
- Előnyök
  - „Automatikusan bővíti, ha betelne”
    - Megszűnéskor a memóriát automatikus felszabadítja
  - Biztonságos hozzáférés az elemekhez
    - Index ellenőrzése

Példa - demo 1

```
int main() {
  vector<Macska> macskak;

  // beszuras a vegere
  Macska cirmos("cirmos");
  macskak.push_back(cirmos);

  // beszuras a vegere
  Macska foltos("foltos");
  macskak.push_back(foltos);

  // vector bejarasa, elemek kiirasa
  for (size_t i = 0; i < macskak.size(); ++i)
    cout << macskak[i].getSzin() << endl;

  // az elso macska "kicserelese"
  Macska szurke("szurke");
  macskak[0] = szurke;

  for (size_t i = 0; i < macskak.size(); ++i)
    cout << macskak[i].getSzin() << ", ";
  cout << endl;
}
```

- Kezdetben egy üres vektor jön létre
- Az elemeket a végére szúrjuk be
  - **Lemásolja**
- [] segítségével elérjük az elemeket
  - 0-tól indexelünk
- `size()`: elemek száma
- Eredmény:
  `cirmos, foltos,`
  `szurke, foltos,`

Létrehozás, inicializálás, értékadás - demo 2

- Üres vektor létrehozása

```
vector<Macska> ures;
```

- 5 elemű vektor
  - Az elemek a default konstruktorral lesznek inicializálva
    - Ennek megléte szükséges

```
vector<Macska> macskak(5);
```

- Inicializálva

```
Macska cirmos("cirmos"), foltos("foltos"), szurke("szurke");
vector<Macska> macskak{cirmos, foltos, szurke};
```

  - Nem is kell külön objektumokat létrehozni

```
vector<Macska> macskak{Macska("cirmos"), {}, {"foltos"}, {"szurke"}};
```

- Egyik vektort értékül adhatjuk a másiknak

```
vector<Macska> uj;
uj = macskak;
```

Elemek hozzáadása, törlése - demo 2

- Vektor esetében a végére lehet hatékonyan beszúrni

```
Macska cirmos("cirmos");
macskak.push_back(cirmos);
macskak.push_back(Macska{"foltos"});
macskak.push_back({"szurke"});
```

  - Eredmény: `cirmos, foltos, szurke`
- Lehetőség van adott elem elé is beszúrni
  - Ehhez egy iterátort kell megadni, ami elé beszúrunk

```
macskak.insert(macskak.begin(), {"feher"});
```

  - Eredmény: **`feher`**`, cirmos, foltos, szurke`
- Utolsó elem törlése

```
macskak.pop_back();
```

  - Eredmény: `feher, cirmos, foltos`
- Egy adott elem törlése

```
macskak.erase(macskak.begin());
```

  - Eredmény: `cirmos, foltos`

Elemek elérése - demo 3

- Lehet indexelni a vektort
  - Ahogy egy tömböt indexelnénk
  - n elem esetén 0-tól n-1-ig

```
vector<Macska> macskak{{"cirmos"}, {"foltos"}, {"szurke"}};
macskak[0] = Macska("feher");
for (size_t i = 0; i < macskak.size(); ++i)
  cout << macskak[i].getSzin() << endl;
```

- **Nincs ellenőrzés** (undefined behavior)

```
macskak[3] = Macska("fekete");
```

- `at(size_t)` i-edik elem „biztonságos” elérése

```
try {
  macskak.at(3) = {"feher"};
  macskak.at(3).kiir();
} catch (const out_of_range& e) {
  cout << "Hiba: rossz index" << endl;
}
```

Elemek bejárása iterátorral - demo 4

- Iterátor használatával

```
for (vector<Macska>::iterator it = macskak.begin(); it != macskak.end(); ++it) {
  cout << it->getSzin() << " -> ";
  it->setSzin("fekete");
  it->kiir();
}
```

- Konstans `vector`: `const_iterator()`, `cbegin()`, `cend()`

```
const vector<Macska> cm(macskak);
for (vector<Macska>::const_iterator cit = cm.cbegin(); cit != cm.cend(); ++cit)
  cit->kiir();
```

  - Nem const iterátor nem is használható

```
for (vector<Macska>::iterator cit = cm.begin(); cit != cm.end(); ++cit)
  cit->kiir();
```

- Visszafelé is be lehet járni

```
for (vector<Macska>::const_reverse_iterator crit = cm.crbegin();
                                            crit != cm.crend(); ++crit)
  crit->kiir();
```

- `auto` használata

```
for (auto crit = cm.crbegin(); crit != cm.crend(); ++crit) ...
```

Elemek bejárása range-based for loop alkalmazásával

- „Egyszerűbb” bejárás
  - Érték szerint járjuk be

```
vector<Macska> macskak(...);
for (Macska m : macskak)
  m.kiir();
```

```
const vector<Macska> cmacskak(...);
for (Macska m : cmacskak)
  m.kiir();
```

- Referencia szerint

```
for (Macska& m : macskak)
  m.setSzin("fekete");
```

```
for (Macska& m : cmacskak)
  m.kiir();
```

- Konstans referencia szerint
  - Nem másolja le (feleslegesen) az elemeket

```
for (const Macska& m : macskak)
  m.kiir();
```

```
for (const Macska& m : cmacskak)
  m.kiir();
```

  - De nem módosíthatjuk

```
for (const Macska& m : macskak)
  m.setSzin("fekete");
```

```
for (const Macska& m : cmacskak)
  m.setSzin("fekete");
```

Macska menhely - demo 5, macska_menhely

Mi a hiba?

```
class Menhely {
  vector<Macska> macskak;
public:
  void befogad(const Macska& m) {
    macskak.push_back(m);
  }

  const Macska& megtekint(size_t i) const {
    return macskak.at(i);
  }

  void kiir() const {
    for (const auto& m : macskak)
      m.kiir();
    cout << endl;
  }
};

int main() {
  Menhely m;
  m.befogad({"cirmi"});
  m.befogad({"foltos"});

  const Macska& elso = m.megtekint(0);
  cout << "megtekijuk a macskat: " << elso.getSzin() << endl;

  m.befogad({"szurke"});
  cout << "ismet megtekijuk a macskat: " << elso.getSzin() << endl;
}
```

Invalidálás

- Amikor változik a vektor mérete, akkor „mozoghatnak az elemek”
  - A fenntartott memóriaterület „betelt”, de új elemet kell beszúrni
    - A `vector` foglal egy nagyobb területet
    - Beszúrja az új elemet
    - Átmásolja (átmozgatja) az elemeket
- Mi okozhatja ezt?
  - Beszúrás
    - `push_back`, `emplace_back`
  - Törlés
    - `erase`, `pop_back`, `clear`
  - Értékadás
    - `v1 = v2`
- Következmény
  - **Iterátorok, referenciák érvénytelenné válnak**

```
auto it = macskak.begin();
Macska& elso = macskak[0];
```

Elemek „mozgatása”

Létrehozás, beszúrás

- Tárolt elemek száma: `size()`
  - Hány elem van eltárolva a vektorban
- Kapacitás: `capacity()`
  - Hány elemnek van memória foglalva
- Kiíratás

```
cout << "meret: " << macskak.size() <<
  ", kapacitas: " << macskak.capacity();
```

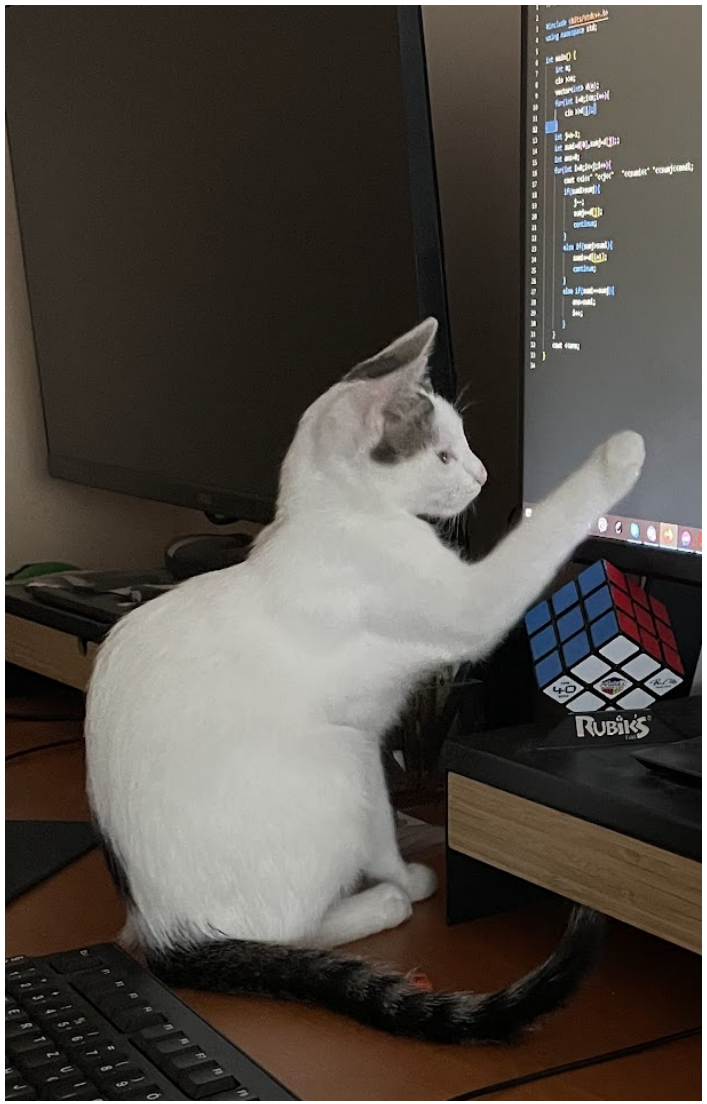
- Vektor létrehozása üresen

```
vector<Macska> macskak;
```

  - `Meret: 0, kapacitas: 0`
- Vektor létrehozása 2 elemmel

```
vector<Macska> mk({"cirmos"}, {"foltos"});
```

  - `Meret: 2, kapacitas: 2`

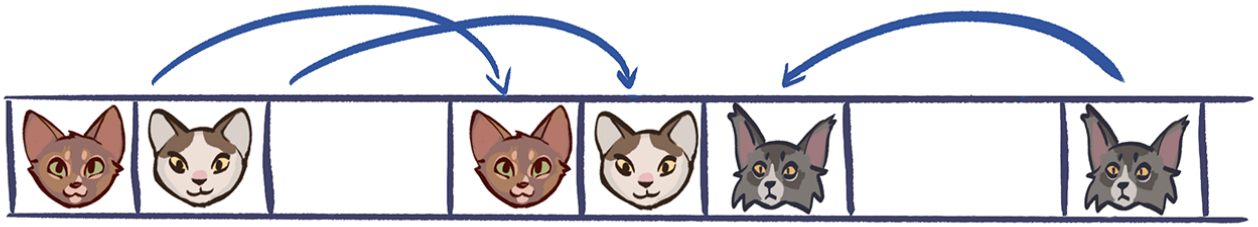


Elemek beszúrása - demo 5 (ismét)

- Van egy 2 elemű vektor, és egy macskát szeretnénk beszúrni

```
vector<Macska> macskak({"cirmos"},{"foltos"});
Macska szurke("szurke");
macskak.push_back(szurke);
```

- A vektor tele van, ezért
  - Új memóriaterületet foglal
  - Beszúrja az új elemet
  - Átmásolja a régi elemeket



- Megszünteti a „régi” elemeket

Előre „foglalunk” helyet az elemeknek - demo 6

Megspóroljuk a bővítést

- 5 macskának „foglalunk” helyet
  - Lefut a default konstruktor 5 alkalommal

```
vector<Macska> macskak(5);
```

- Majd beállítjuk az első 3 elemet
  - Nem `push_back`

```
macskak[0] = {"cirmos"};
macskak[1] = {"foltos"};
macskak[2] = {"szurke"};
```

Előre foglalunk memóriát - demo 7

Memória foglalás

- Nem kellenek az elemek, csak memóriát akarunk foglalni
  - Csak a kapacitás változik (5), a vektor üres (méret: 0)
  - Nem jönnek létre macska objektumok
    - Lefoglal `5*size_of(Macska)` méretű területet

```
vector<Macska> macskak;
macskak.reserve(5);
```

Előre foglalunk memóriát

Elemek beszúrása

- Macska beszúrása

```
macskak.push_back({"cirmos"});
```

  - Egy (ideiglenes) macska objektum létrejön
  - Lemásolja a vektor magának
    - *Átmozgatja, ha tudja*
  - Az (ideiglenes) macska objektum megszűnik

```
macskak.push_back({"foltos"});
macskak.push_back({"szurke"});
```

- `Méret: 3, kapacitás: 5`

Ott létrehozni, ahol használjuk

- Van lehetőség arra is, hogy a „felhasználás helyén” hozzunk létre objektumot
  - Nem jön létre feleslegesen (ideiglenes objektum)
    - Ami lemásolódik majd megszűnik

```
macskak.push_back(Macska{"foltos", 2024});
```

- `emplace_back`
  - A vektor végén hozza létre az elemet
  - Nem az objektumot kell átadni, hanem a konstruktor argumentumait

```
macskak.emplace_back("szurke", 2024);
```

range-based for loop

Mi is ez valójában?

- Hogyan működik valójában?

```
for (Macska& m : macskak)
  cout << m.getSzin() << endl;
```

  - Indexelés?
  - Iterátor?
  - Valami más?

- `https://cppinsights.io/`

set

- `https://en.cppreference.com/w/cpp/container/set`
- `set<T> v;`
  - `T` a típus
- `T` elemekből álló halmazt létre
  - Az elemszámra nincs felső korlát
    - Csak a memória mérete
- Az elemeket rendezetten tárolja
  - Szükséges, hogy az elemeket össze lehessen hasonlítani
    - A $<$ operátort használja
  - Az elemeket egy bináris fában tárolja
    - Implementáció függő

Példa

```
int main() {
  vector<int> v {1, 5, 7, 2, 7, 1, 5};
  set<int> s    {1, 5, 7, 2, 7, 1, 5};

  for (int i : v)
    cout << i << " ";
  cout << endl;       // 1 5 7 2 7 1 5

  for (int i : s)
    cout << i << " ";
  cout << endl;       // 1 2 5 7

  set<int>::iterator it = s.find(2);
  if (it != s.end())
    cout << "2 benne van" << endl;

  s.erase(it);
  for (auto it = s.begin();
            it != s.end(); ++it)
    cout << *it << " ";
  cout << endl;       // 1 5 7

  s.erase(1);
  for (int i : s)
    cout << i << " ";
  cout << endl;       // 5 7
}
```

- Létrejön egy vektor (vector) és egy halmaz (set)
  - Ugyanazokkal az elemekkel inicializáljuk
- Bejárások
  - Iterator
  - Range based for
- Halmaz elemei
  - „Azonos elemből” csak egyet tárol
- Keresés: `find`
  - Iterátort ad vissza
- Törlés
  - `erase`
    - Iterator
    - Elem

Létrehozás, beszúrás

- Üres halmaz létrehozása

```
set<int> halmaz;
```

- Kezdeti értékek megadása

```
set<int> s1 { 1, 5, 7, 2, 7, 1, 5, 2 };
set<int> s2 (v.begin(), v.end());
set<int> s3 (v.begin(), v.begin()+3);
```

- Beszúrás
  - Nem a végére vagy az elejére szúrjuk be
  - A halmaz dönti el, hogy hova kerül az elem
    - Rendezve tárolja az elemeket
    - Ha már volt, akkor be sem szúrja
  - A beszúrás visszaadja, hogy sikerült-e, illetve egy iterátort a tárolt elemre

```
pair<set<int>::iterator, bool> beszur = s.insert(6);
if (beszur.second)
  cout << "sikerult beszurni: " << *beszur.first << endl;
else
  cout << "mar benne volt: " << *beszur.first << endl;
```

Keresés, törlés

- Keresés
  - find: visszaad egy iterátort az adott elemre
    - set::end - ha nincs benne

```
set<int>::iterator keres = s.find(6);
if (keres != s.end())
  cout << "benne van: " << *keres << endl;
```

  - contains: megmondja, hogy az adott eleme benne van-e (C++20)

```
if (s.contains(6))
  cout << "a 6 benne van" << endl;
```

  - `lower_bound(kulcs)`
    - iterátor az első olyan elemre, amelyik nem kisebb az adott elemnél
  - `upper_bound(kulcs)`
    - iterátor az első olyan elemre, amelyik nagyobb az adott elemnél
- Törlés
  - Iterátor alapján

```
s.erase(keres);
```

  - Érték alapján

```
s.erase(5);
```

Bejárások

- Nincs index operátor
- Iterátorral

```
for (set<int>::iterator it = s.begin(); it != s.end(); ++it)
  cout << *it << " ";
```

  - Hasonlóan van const és reverse iterátor is

- Ranged-based for ciklus

```
for (int i : s)
  cout << i << " ";
```

Invalidálás

- Az elemeket egy fában tárolja
  - Egy új eleme beszúrásakor csak az új elemnek kell memóriát foglalni
    - A többi elem marad a helyén a memóriában
  - Egy meglévő csúcs alá lesz bekötve (és a fa átrendeződik)
    - Az érintett csúcsok élei megváltoznak
- Törlés esetén az adott elem törlődik
  - A fa élei változnak, de a többi elem ugyanott marad a memóriában
- Invalidálás
  - Csak a törölt elemen álló iterátor és az elem referenciái lesznek invalidak
    - A többi iterátor és referencia használható

```
void mitCsinal(set<int>& s) {
  for (auto it = s.begin(); it != s.end(); ++it)
    if (auto kov = it; ++kov != s.end())
      s.erase(kov);
}
```

map

- `https://en.cppreference.com/w/cpp/container/map`
- `map<K, E> m;`
  - `K` és `E` típusok
- `K` és `E` típusú párokból álló halmazt létre
  - Az elemszámra nincs felső korlát
    - Csak a memória mérete
- Kulcs-érték párokat tárol
  - Kulcs szerint rendezetten tárolja az elemeket
    - Két azonos kulcsú elemet nem tárol
  - Szükséges, hogy a kulcs elemeket (`K`) össze lehessen hasonlítani
    - A $<$ operátort használja
    - Az elemeket egy bináris fában tárolja
  - Az értékre (`E`) nincs megkötés

Példa

```
int main() {
  map<string, int> m { {"Cirmi", 2024}, {"Garfield", 1976} };

  for (auto it = m.begin(); it != m.end(); ++it)        // Cirmi 2024
    cout << it->first << " " << it->second << endl;     // Garfield 1976

  pair<map<string, int>::iterator, bool> beszur =
    m.insert( {string("Cirmi"), 2023} );

  if (beszur.second)
    cout << "mar benne volt: " << beszur.first->second << endl;
  else
    cout << "nem volt benne: " << beszur.first->second << endl;

  for (const pair<string, int>& p : m)                  // Cirmi 2024
    cout << p.first << " " << p.second << endl;         // Garfield 1976

  m["Mirr-Murr"] = 2010;
  for (const auto& p : m)                               // Cirmi 2024
    cout << p.first << " " << p.second << endl;         // Garfield 1976
                                                        // Mirr-Murr 2010

  m["Mirr-Murr"] = 2020;
  for (const auto& [kulcs, ertek]  : m)                 // Cirmi 2024
    cout << kulcs << " " << ertek << endl;              // Garfield 1976
                                                        // Mirr-Murr 2020
                                                        //           ^^^^ megvaltozott
}
```

Létrehozás, beszúrás, keresés, törlés, bejárás

- A `map` és a `set` esetében „azonosak a műveletek”
  - Létrehozás, beszúrás, keresés, törlés, bejárás
- `map` esetében van indexer operátor
  - A kulcsot lehet vele indexelni
    - `m["Mirr-Murr"] = 2010;`
  - Ha már benne volt az adott kulcs, akkor egy referenciád ad az értékre
    - Ilyenkor *Mirr-Murr* értékát átírjuk 2010-ra
  - Ha még nem volt benne, akkor beszúrja a kulcs-érték párt
    - Az értéket a default konstruktorral inicializálja
    - Visszaad egy referenciát, és az módosíthatom (pl. 2010-re állítom)
    - Ez az eset „majdnem olyan”, mintha beszúrtam volna
- structured binding (C++17)

```
for (const auto& [kulcs, ertek]  : m)
  cout << kulcs << " " << ertek << endl;
```

Tartalom

Operátor overloading

Motiváció

- Miért legyenek különböző nevű függvények, ha ugyanazt csinálják
  - Például a maximum egész és valós esetében is ugyanazt jelenti
    - Mind a két esetben ugyanazt a nevet adnánk a függvénynek

```
int    max(int a, int b) { /* ... */ }
double max(double a, double b) { /* ... */ }
int    max(int a, int b, int c) { /* ... */ }
```

- Konstruktor esetében nem is lehet másik nevet adni

```
class Macska {
  /* ... */
public:
  Macska() { /* ... */ }
  Macska(const string& nev) { /* ... */ }
  Macska(const string& nev, unsigned szulEv) { /* ... */ }
};
```

- `const` és nem `const` metódusoknak is lehet ugyanaz a neve

```
class Macska {
  /* ... */
public:
  string& getNev() { return nev; }
  const string& getNev() const { return nev; }
};
```

Egy kis matematika ...

Komplex számok - nem kell tudni a műveleteket

- A komplex számok a valós számok bővítése
  - El lehet végezni a negatív számból való gyökvonást
    - i-vel jelölik a képzetes egységelemet ($i^2 = -1$)
  - Minden komplex szám $v + ki$ alakban írható fel
    - v a valós (real) rész
    - k a képzetes (imaginary) rész
- Komplex számokon értelmezett műveletek
  - Összeadás: $(v+ki) + (w+li) = (v+w) + (b+d)i$
  - Kivonás: $(v+ki)$ - $(w+li) = (v-w) + (k-l)i$
  - Szorzás: $(v+ki)$ * $(w+li) = (vw-kl) + (vl+kw)i$
  - Szorzás skalárral: s * $(v+ki) = (sv) + (sk)i$
- Példa
  - $5 + 5 = 10$
  - $(3 + 2i) + (4 - i) = 7 + i$
  - $(3 + 2i) + (-3 + 5i) = 7i$
  - $(3 + 2i)$ * $(3 - 2i) = 13$

Complex szám implementáció

```
class Compl {
  int v, k; // valos, kepzetes
public:
  Compl(int v = 0, int k = 0) :
    v(v), k(k) { }

  Compl szorzas(const Compl& c) const {
    Compl res;
    res.v = v * c.v - k * c.k;
    res.k = v * c.k + k * c.v;
    return res;
  }

  Compl plusz(const Compl& c) const {
    Compl res (v + c.v, k + c.k);
    return res;
  }

  Compl szorzas(int s) const {
    return Compl(s*v, s*k);
  }

  string text() const {
    if (k == 0)
      return to_string(v);
    else
      return "(" + to_string(v) +
        " + " + to_string(k) + "i)";
  }
};
```

- Hogyan használjuk
  - $5 + 6 = 11$
  - $(3 + 2i) + (4 - i) = 7 + i$
  - $(3 + 2i) * (3 - 2i) = 13$

```
int main() {
  int i = 5, j = 6;
  cout << i << " + " << j << " = "
       << i+j << endl;

  Compl c1(3, 2), c2(4, -1), c3(3, -2);
  Compl c = c1.plusz(c2);

  cout << c1.text() << " + "
       << c2.text() << " = "
       << c.text() << endl;

  cout << c1.text() << " * "
       << c3.text() << " = "
       << c1.szorzas(c3).text() << endl;
}
```

Mennyire olvasható?

- `((3 + a) * (b + c) / d + 5) * (e + a * (2 * (f + 3)))`
  - Ha a, b, c, d, e, f egész számok, akkor jól olvasható a kód

```
((3 + a) * (b + c) / d + 5) * (e + a * (2 * (f + 3)))
```

  - Mi van akkor, ha ezek komplex számok
    - A „3 + a” eleve gondot okozna ☺

```
a.plusz(3).szorzas(b.plusz(c)).osztas(d).plusz(5).szorzas(
  e.plusz(a.szorzas(f.plusz(3).szorzas(2)))
)
```

- Ha az egész/valós számok esetében írhatok matematikai műveleteket,
  - akkor a komplex számokhoz miért írjak „verset”?
- Melyik olvashatóbb?

```
str = "nev: " + cirmos.getNev() + ", szul ev:" + to_string(cirmos.getSzulEv());
cout << i << " + " << j << " = " << i+j << endl;
```

```
str = string("nev: ").append(cirmos.getNev()).append(", szul ev:").
        append(to_string(cirmos.getSzulEv()));
cout.kiir(i).kiir(" + ").kir(j).kiir(" = ").kiir(i+j).sortores();
```

Írjunk mi is (valódi) operátorokat

```
class Compl {
  int v, k; // valos, kepzetes
public:
  Compl(int v = 0, int k = 0) :
    v(v), k(k) { }

  Compl plusz(const Compl& c) const {
    return { v + c.v, k + c.k };
  }

  Compl szorzas(const Compl& c) const {
    return { v * c.v - k * c.k,
             v * c.k + k * c.v };
  }

  Compl szorzas(int s) const {
    return { s * v, s * k };
  }

  string text() const {
    if (k == 0)
      return to_string(v);
    else
      return "(" + to_string(v) + " + "
             + to_string(k) + "i)";
  }
};
```

```
class Compl {
  int v, k; // valos, kepzetes
public:
  Compl(int v = 0, int k = 0) :
    v(v), k(k) { }

  Compl operator+(const Compl& c) const {
    return { v + c.v, k + c.k };
  }

  Compl operator*(const Compl& c) const {
    return { v * c.v - k * c.k,
             v * c.k + k * c.v };
  }

  Compl operator*(int s) const {
    return { s * v, s * k };
  }

  operator string() const {
    if (k == 0)
      return to_string(v);
    else
      return "(" + to_string(v) + " + "
             + to_string(k) + "i)";
  }
};
```

```
Compl c = c1.plusz(c2);
cout << c1.szorzas(c3).text() << endl;
```

```
Compl c = c1 + c2 * c3;
cout << (string)c << endl; // lesz szebb
```

C++ operátorok

Prec.	Operátor	Leírás	Asszoc.
1	::	**Scope**	→
2	a++ a−−	Postfix inc./dec.	
	type() type{}	Függvény konverzió	
	a ( )	Függvény hívás	→
	a [ ]	Index	
	. −>	Member hozzáférés	
3	++a −−a	Prefix inc./dec.	
	+a −a	Unáris plusz, minusz	
	! ~	Logikai NOT és bitenkénti NOT	
	(type)	C-stílusú típuskoverzió	
	*a	Dereferencia	←
	&a	Address-of	
	`sizeof`	Sizeof	
	`new new` [ ]	**Dinamikus memória foglalás**	
	`delete delete` [ ]	**Dinamikus memória felszabadítás**	
4	.* ->*	Pointer-to-member	→
5	a*b a/b a%b	Szorzás, osztás, maradék	→
6	a+b a−b	Összeadás, kivonás	→

C++ operátorok (folyt.)

Prec.	Operátor	Leírás	Asszoc.
7	<< >>	Bitenkénti balra és jobbra léptetés	→
8	<=>	**Threeway comp. (C++20)**	→
9	< <= > >=	< és ≤ reláció > és ≥ reláció	→ →
10	== !=	= és ≠ reláció	→
11	&	Bitenkénti ÉS	→
12	^	Bitenkénti KIZÁRÓ VAGY	→
13	|	Bitenkénti VAGY	→
14	&&	Logikai ÉS	→
15	||	Logikai VAGY	→
16	a?b:c throw = += -= *= /= %= <<= >>= &= ^= |=	Feltételes **Kivétel dobás** Értékadás Összetett értékadás (+, -) Összetett értékadás (*, /, %) Összetett értékadás (bit léptetés) Összetett értékadás (bit műveletek)	←
17	,	Vessző (szekvencia)	→

C++ operátor-kiterjesztés

- A metódushoz hasonlóan az operátorokat is felül lehet definiálni
  - Miért kelljen más nevet adni a saját típusok hasonló műveleteinek
- A következő operátorok definiálhatóak felül
  - Aritmetikai, bináris, értékadó, összehasonlító és logikai operátor
  - * (dereferencia), ->, (), [], vessző operátor és a pointer to member
  - `new`, `new[]`, `delete`, `delete[]`
  - Típuskonverzió
- Megszorítások
  - Az operátor prioritását vagy asszociációt nem lehet megváltoztatni
  - A paraméterek számát nem lehet megváltoztatni
  - A && és || operátorok elveszítik a rövidített kiértékelésüket
    - `if (p && 3 < p->i) { /* ... */ }`
- **típus** `operator` **operátor** (par1, ...) `{ /* ... */ }`
  - típus: az operátor visszatérési típusa
  - `operator` ezt a kulcsszót kell használni
  - operátor: az az operátor, aminek új jelentést akarunk adni
  - par1, ...: paraméterek (amennyi az adott operátorhoz kell)

Aritmetikai operátorok

„Klasszikus megvalósítás”

- Összeadás: `a + b`
  - **Új objektum jön létre**
    - a+b értékét tárolja
    - a és b értéke **nem változik**
- Értékadás növeléssel: `a += b`
  - a értékét megnöveljük b-vel
    - b értéke **nem változik**
    - a értéke **megváltozik**
  - **A kifejezés értéke** a
    - Nem jön létre új objektum

```
class Compl {
  int v, k; // valos, kepzetes
public:
  Compl(int v=0, int k=0) : v(v), k(k) {}

  Compl operator+(const Compl& c) const {
    return { v + c.v, k + c.k };
  }

  Compl operator+(int s) {
    return { v + s, k };
  }

  Compl& operator+=(const Compl& c) {
    v += c.v;
    k += c.k;
    return *this;
  }

  Compl& operator+=(int s) {
    v += s;
    return *this;
  }
};

c1 += 3;
c = c1 + c2;
c += c3;
```

Globális operátor

- Ha egy osztály metódusa egy operátor, akkor az operátor bal oldali (vagy az egyetlen) operandusa az adott osztály
  - Amit megvalósítottunk
    - Compl + Compl
    - Compl + egész
  - Egész + egész
    - Ez adott

```
class Compl {
  int v, k; // valos, kepzetes
public:
  Compl operator+(const Compl& c) const {
    return { v + c.v, k + c.k };
  }

  Compl operator+(int s) const {
    return { v + s, k };
  }
};
```

- Ha nem az adott osztály típusa a bal oldali operandus, akkor nem lehet az osztály metódusa
  - Globális operátorral meg lehet oldani
    - Lehet **friend** függvény is

```
class Compl {
  /* ... */
  friend Compl operator+(int s, const Compl& c) {
    return { s + c.v, c.k };
  }
};
```

Pre és post növelő operátor

„Klasszikus megvalósítás”

- Pre növelő operátor: ++i;
  - „Megnöveli” az objektumot
  - Visszaadja a megnövelt objektumot
    - Kifejezés értéke ez lesz

```
Compl c(5, 1);
(++c).kiir(); // 6 + 5i
c.kiir();     // 6 + 5i
```

- Post növelő operátor: i++;
  - „Megnöveli” az objektumot
  - Az eredeti objektumot adja vissza
    - A kifejezés rétéke ez lesz

```
Compl c(5, 1);
(c++).kiir(); // 5 + 5i
c.kiir();     // 6 + 5i
```

```
class Compl {
  int v, k;
public:
  //valos reszt noveli
  Compl& operator++() {
    ++v;
    return *this;
  }

  // valos reszt noveli
  Compl operator++(int) {
    // obj. "elmentese"
    Compl tmp = *this;
    // obj. "novelese"
    operator++();
    // eredeti vosszaadasa
    return tmp;
  }

  void kiir() const {
    cout << v << " + " <<
         << k << "i" << endl;
  }
};
```

Konverziós operátor

- A visszatérési értéke adott
  - Az a típus, amire konvertálunk
    - Nem is lehet megadni/kiírni
- Ez is egy operátor
  - Lehet `const` is
  - Tetszőleges megvalósítás

```
class Compl {
  operator string() const {
    return "(" + to_string(v) +
       " + " + to_string(k) + "i)";
  }

  operator int() const {
    return v;
  }
};

int main() {
  Compl c(5, 1);
  int i = c;      // i = 5
  string s = c;   // s = "(5 + 1i)"
}
```

- Van `Compl`-ról `string`-re konv.
  - A függvényhívásnál automatikusan végrehajtódik a konverzió

```
void kiir(const string& s) {
  cout << s << endl;
}

int main() {
  Compl c(5, 1);
  kiir(c);
}
```

Típuskonverzió

Mellékhatás

- Nem az a függvény vagy metódus hívódik, amit gondolunk
  - A paraméter típusra van automatikus konverzió
    - De így mást jelent, mint amit szeretnénk

```
class Compl {
public:
  Compl(int v) : v(v), k(0) { }

  operator string() const { /* ... */ }
};

int abs_ertek(int n) {
  if (n < 0)
    return -n;
  return n;
}

string toString(const Compl& c) {
  return to_string(c.v) + " + " + to_string(c.k) + "i";
}

int main() {
  Compl c(-5, 1);
  cout << abs_ertek(c) << endl; // 5, ami NEM c abs. erteke
  cout << toString(11) << endl; // "11 + 0i", de csak a 11-et akartuk latni
}
```

Típuskonverzió

explicit

- Ha meg akarjuk tiltani az automatikus típus-konverziót, akkor `explicit`-té kell tenni az operátort vagy a konstruktort
  - Ilyenkor ki kell írni, ha konvertálni akarunk

```
class Compl {
public:
  explicit Compl(int v) : v(v), k(0) { }

  explicit operator string() const { /* ... */ }
};

int abs_ertek(int n);

string toString(const Compl& c);

int main() {
  Compl c(-5, 1);
  cout << abs_ertek((int)c) << endl;   // 5, "nem jo", de szandekosan csinaltuk
  cout << toString((Compl)11) << endl; // "11 + 0i", most a komplex 11-et latjuk
}
```

```
int main() {
  Compl c(-5, 1);
  cout << abs_ertek(c) << endl; // FORDITASI HIBA
  cout << toString(11) << endl; // FORDITASI HIBA
}
```

Kiíratás megvalósítása

- A `cout` egy globális objektum, aminek a típusa `ostream`
  - Meg van valósítva a << operátor az `ostream`-re és a C++ típusokra
- Mi is megvalósíthatjuk a saját osztályunkra
  - Nem lehet metódus, mert a jobb oldali operandus az osztályunk
    - `friend`-ként érdemes megvalósítani
  - Érdemes visszaadni az ostream-et
    - Ekkor használhatjuk egymás után többször `cout << c1 << c2;`

```
class Compl {
  int v, k; // valos, kepzetes
public:
  /* ... */

  friend ostream& operator<<(ostream& os, const Compl& c) {
    os << "(" << c.v << " + " << c.k << "i)";
    return os;
  }
};

int main() {
  Compl c1(1, 2), c2(3, 4);
  cout << c1 << " + "  << c2 << " = " << c1+c2 << endl;
}
```

Operátor-kiterjesztés kérdések

- Mi definiáljuk, hogy hogyan működik, de ...
- Mivel térjünk vissza?
  - `void` vagy adjunk vissza egy objektumot?
  - Érték, referencia, konstans referencia, (pointer)?
- Mi legyen a paraméter?
  - A típus „adott”
  - Érték, referencia, konstans referencia, (pointer)?
- Megváltozik az objektum?
  - Az operátor `const` legyen?
- Mit csináljon az operátor?
  - Ez általában adott, de ...
    - Hibákat hogyan kezeljük?
- Gondoltunk a melléhatásokra?
  - És a felhasználók gondolnak majd?
- Megnéztük, hogy „eredetileg” hogyan működnek?
  - Annak megfelelően érdemes csinálni

Tartalom

Újrafelhasználás

- A objektum-orientál programozás egyik alapelve az újrafelhasználás
- Aggregáció, kompozíció
  - Rész-egész kapcsolat
  - Az újrafelhasználás során „módosítás nélkül” felhasználunk egy már meglévő típus
- Öröklődés
  - Az újrafelhasználás egy másik formája
  - A meglévő típust specializáljuk
    - Kiegészíthetjük újabb tulajdonságokkal és operációkkal
    - Meglévő operációkat felüldefiniálhatjuk

Öröklődés

- Az öröklődés megadása
  - Az osztály deklaráció után kettőspont
  - Az öröklődés láthatósága
    - `public`, `protected` vagy `private`
    - Nem kötelező megadni (de érdemes kiírni)
    - Ha nem adjuk meg, akkor `private` lesz
  - Ősosztály neve
    - Ebből származik az osztály
    - Örökli metódusait és attribútumait
    - És minden egyebet (pl. típus definíció)

```
class Emlos {
  /* ... */
};

class Macska : public Emlos {
  /* ... */
};
```

Öröklődés

Példa (az inicializálás nem teljes!)

```
class Emlos {
  unsigned szulEv;
public:
  Emlos(unsigned szulEv = 2023) : szulEv(szulEv) {
  }

  unsigned getSzulEv() const {
    return szulEv;
  }
};

class Macska : public Emlos { // oroklodes megadasa
  string nev;
public:
  Macska(unsigned szulEv, const string& nev) : nev(nev) {
  }

  const string& getNev() const {
    return nev;
  }
};

int main() {
  Macska macska(2021, "Cirmi");
  // macska.getSzulEv() orokolt metodus hivasa
  cout << macska.getSzulEv() << " " << macska.getNev() << endl;
}
```

Öröklődés láthatósága

- Megadhatjuk az öröklődés láthatóságát
  - Ha nem adjuk meg, akkor `private` lesz
- `public` öröklődés
  - Az örökölt tagok láthatósága nem változik
    - Ami `public` volt az `public` marad,
    - Ami `protected` volt az `protected` marad,
    - Ami `private` volt az ősosztályban, az is része lesz a leszármazott osztálynak, de nem lesz elérhető
- `protected` öröklődés
  - A `public` láthatóságú tagok láthatósága `protected`-re csökken
    - Ami `protected` volt, annak a láthatósága nem változik
  - Ami `private` volt az ősosztályban, az is része lesz a leszármazott osztálynak, de nem lesz elérhető
- `private` öröklődés
  - A `public` és `protected` örökölt tag `private` láthatóságúra változik
  - Ami `private` volt az ősosztályban, az is része lesz a leszármazott osztálynak, de nem lesz elérhető

Privát tagok öröklése

- Ha egy tag `private` láthatóságú, akkor csak az adott osztály férhet hozzá
  - **Még a leszármazott osztályok sem**
- Ennek ellenére a gyerek osztályok megöröklik a privát tagokat is
  - Adattagok részei lesznek az osztálynak
  - Lesznek olyan metódusai, amelyek az ősosztályban privátok voltak
- Az örökölt tagok „használhatják” ezeket, de nem közvetlenül
  - Ősosztályban lévő metódusok hívhatják egymást
  - Hozzáférhetnek az adattagokhoz

```
class Emlos {
  unsigned szulEv;
public:
  unsigned getSzulEv() const {
    return szulEv;
  }
  /* ... */
};
```

```
class Macska : public Emlos {
public:
  bool isFelnott() const {
    return 3 < 2023 - getSzulEv();
  }
};
```

```
class Macska : public Emlos {
public:
  bool isFelnott() const {
    return 3 < 2023 - szulEv; // HIBA
  }
};
```

Objektum inicializálása

- Az adattagok a deklaráció sorrendjében lesznek inicializálva
  - Az örökölt adattagok hamarabb
  - Az osztály saját adattagjai később
- A konstruktor inicializáló listában lehetőség van meghívni az ősosztály konstruktorát
  - Az ősosztály adattagjait ott nem lehet inicializálni
  - A konstruktor hívást kell elsőnek megadni
    - Nem kötelező, de akkor „nem jó sorrendben” lesznek inicializálva az adattagok

```
class Emlos {
  unsigned szulEv;
public:
  Emlos(unsigned szulEv = 2023) :
    szulEv(szulEv) {
  }
};
```

```
class Macska : public Emlos {
  string nev;
public:
  Macska(unsigned szulEv, const string& nev) :
    Emlos(szulEv), nev(nev) {
  }
};
```

Név ütközés feloldása

```
class Emlos {
protected:
  string nev = "emlos";
public:
};
```

```
class Macska : public Emlos {
  string nev;
public:
  Macska(const string& nev) : nev(nev) {
  }

  void kiir() const {
    cout << nev << endl;
    cout << Emlos::nev << endl;
  }
};
```

- Az ősosztályban és a gyerekosztályban is van azonos nevű tag
  - A `Macska` osztály örökli a `nev` adattagot az `Emlos` osztályból
  - A `Macska` osztályban is definiáltunk egy `nev` adattagot
- Mind a kettő része lesz az osztálynak
  - Két külön adattag
- Hivatkozás
  - `nev` vagy `this->nev`: a gyerekosztályban található `nev`-et jelenti
  - Az örököld adattagra az ősosztály scope-olt nevével hivatkozhatunk
    - `Emlos::nev = " ... ";`
    - `this->Emlos::nev = " ... ";`

Tartalom

Metódus felüldefiniálás

- Az öröklődéssel nem csak új funkcionalitásokat adhatunk az ősosztályhoz, hanem a meglévőket is felül definiálhatjuk
  - Ugyanaz a funkcionalitás, csak a speciális esetben „speciálisabb” jelentése lehet
- A leszármazott osztályban ugyanazzal a szignatúrával kell a megadni a metódust
  - Ha eltérés van, akkor nem lesz felüldefiniálás
    - Két „különböző” metódus lesz
- Típus alapján lesz eldöntve, hogy melyik hívódik
  - Statikus kötés
    - Fordítási időben eldől, hogy melyik metódus hívódik
  - Dinamikus kötés
    - Csak futásidőben derül ki, hogy melyik metódus fog meghívódni

Metódus „felüldefiniálás”

Felüldefiniálás nélkül

```
class Sikidom {
  const string tip;

protected:
  Sikidom(string s) : tip(s) { }

public:
  Sikidom() : tip("Sikidom") { }

  const string& tipus() const {
    return tip;
  }

  double kerulet() const {
    return -1; // hibajelzes, de jobb lenne kivetel vagy letiltas
  }
};

class Haromszog : public Sikidom {
  double a, b, c;

public:
  Haromszog(double a, double b, double c) :
    Sikidom("Haromszog"), a(a), b(b), c(c) { }

  double kerulet() const {
    return a+b+c;
  }
};
```

„Klasszikus” metódus hívás

```
int main() {
  Sikidom s;
  cout << s.tipus() << " ";
  cout << s.kerulet() << endl;
  Haromszog h(3, 4, 5);
  cout << h.tipus() << " ";
  cout << h.kerulet() << endl;
}
```

```
class Sikidom {
  /* ... */
  const string& tipus() const;
  double kerulet() const;
};

class Haromszog:public Sikidom{
  /* ... */
  double kerulet() const;
};
```

- Kimenet: `Sikidom -1`

  `Haromszog 12`
- Mindkét esetben ugyanaz a `tipus()` metódus hívódott meg
  - Példányosításkor más a „tip” értéke, mást ad vissza metódus
- Két `kerulet()` metódusa van a Sikidom osztálynak
  - Statikus típus alapján fordítási időben eldől, hogy melyik hívódik
    - A fordító látja az objektumok pontos típusát
    - A leszármazott osztályban lévő lesz meghívva
    - Akár más is lehetne a metódus neve

Érték szerinti paraméter átadás

- A leszármazott osztályokat kezelhetjük „ősosztály” típusként is
  - Ahol ősosztályt „használunk”, ott használhatjuk a gyerek osztályt is
- Érték szerinti paraméterátadás
  - Azt másolja le, ami megadunk

```
void kiir(Sikidom sikidom) {
  cout << sikidom.tipus() << ": kerulet = " << sikidom.kerulet() << endl;
}

int main() {
  Sikidom sikidom;
  Haromszog haromszog(3, 4, 5);
  szamol(sikidom);      // Sikidom: -1
  szamol(haromszog);    // Haromszog: -1
}
```

- Csak a `Sikidom` részét másolja le a `Haromszog` típusú objektumnak
  - Akármit adunk át, abból „`Sikidom` lesz”

Referencia szerinti paraméterátadás

- Hívásnál nem másolja az objektumot, csak egy referenciát ad át az eredeti objektumra
  - A statikus típusa az ősosztály lesz
  - A dinamikus típusa az eredeti típus (ami lehet leszármazott osztály is)

```
void szamol(const Sikidom& sikidom) {
  cout << sikidom.tipus() << ": " << sikidom.kerulet() << endl;
}

int main() {
  Sikidom sikidom;
  Haromszog haromszog(3, 4, 5);
  szamol(sikidom);      // Sikidom: -1
  szamol(haromszog);    // Haromszog: -1
}
```

- Azt várnánk el, hogy a „megfelelő” `kerulet()` fog meghívódni
  - `Sikidom` esetében -1 lesz az eredmény
  - `Haromszog` esetében 12, **de itt is -1-et kapunk**

Statikus kötés

- Ha egy metódus **nem vitruális**, akkor az **objektum statikus típusa alapján lesz bekötve a metódushívás**
  - Ha az ősosztály típusa alapján használjuk, akkor az ősosztály metódusa lesz bekötve
    - Pointer vagy referencia segítségével tudjuk ősosztály típusúként használni
- Mindegy, hogy `Sikidom` vagy `Haromszog` lesz a paraméter, a `Sikidom::kerulet() const` lesz meghívva
  - A `Haromszog` esetében ez rossz működést eredményez
    - A `kerulet()` metódus fordítási időben „be lesz kötve” a `Sikidom` osztály metódusára

```
void szamol(const Sikidom& sikidom) {
  cout << sikidom.tipus() << ": " << sikidom.kerulet() << endl;
}
```

Polimorfizmus (többalakúság)

- A polimorfizmus az objektumok felcserélhetőségét biztosítja
  - Az előző példában ez nem működött
- Az objektumot az őstípus alapján kezeljük
  - A kód nem függ a specifikus típustól
  - Bármilyen származtatott típus is használható
    - Lehetőségünk van később is definiálni azt
- Az őstípus interfészét használjuk
  - Az általa definiált funkcionalitásokat
- Fordítási időben nem dől el, hogy pontosan melyik metódus hívódik
  - Csak futás közben derül ki
    - Szemben a statikus kötéssel

Virtuális metódus

- A polimorfizmushoz szükséges, hogy ne fordítási időben legyenek bekötve a hívások
  - Ehhez virtuálissá kell tenni a metódusokat
    - C++ esetében a metódusok nem virtuálisok
- `virtual` kulcsszóval tehetjük meg
  - A metódus elé kell kiírni

```
class Sikidom {
  /* ... */
  virtual double kerulet() const {
    return -1;
  }
};
class Haromszog : public Sikidom {
  /* ... */
  virtual double kerulet() const {
    return a+b+c;
  }
};
```

Virtuális metódus

Példa

```
class Sikidom {
  /* ... */
  virtual double kerulet() const {
    return -1;
  }
};

class Haromszog : public Sikidom {
  /* ... */
  virtual double kerulet() const {
    return a+b+c;
  }
};

void szamol(const Sikidom& sikidom) {
  cout << sikidom.tipus() << ": " << sikidom.kerulet() << endl;
}

int main() {
  Sikidom sikidom;
  Haromszog haromszog(3, 4, 5);
  szamol(sikidom);       // Sikidom: -1
  szamol(haromszog);     // Haromszog: 12
}
```

Virtuális metódus

- Ha egy metódus egyszer virtuális lett, akkor az is marad
  - Ha felüldefiniáljuk, de nem írjuk ki a felüldefiniálásnál a `virtual` kulcsszót, akkor is virtuális lesz
- A `Haromszog` osztály `kerulet()` `const` metódusa virtuális lesz, mert az `Sikidom` osztályban ez a metódus virtuális
  - Akkor is, ha nincs kiírva elé a `virtual` kulcsszó

```
class Sikidom {
  /* ... */
  virtual double kerulet() const;
};

class Haromszog : public Sikidom {
  /* ... */
  double kerulet() const; // virtualis!!!
};
```

override

- Felüldefiniálnám, de ...
  - Már az ősosztályban lehagytam a `virtual` kulcsszót
  - Elrontottam a signatúrát
    - Például a `const` vagy referencia lemaradt

```
class Os {
public:
  virtual void f() const;

  virtual void g(string& s);

  virtual void h(const char*);

  virtual void IoT();

  void ok();
};
```

```
class Gyerek : public Os {
public:
  void f();          // const

  void g(string s); // &

  void h(char*);     // const

  void loT();        // eliras

  void ok();         // nem virt.
};
```

- Egyik esetben sincs felüldefiniálás
  - Nem okoz fordítási hibát

override (folyt.)

- Az ilyen hibák kivédésére használhatjuk az `override` azonosítót
  - A metódusdeklaráció végére kell írni
- Ha nem definiál felül metódust, amire alkalmazzuk, akkor fordítási hibát kapunk

```
class Sikidom {
public:
  virtual double kerulet() const;
  const string& tipus() const;
};

class Haromszog : public Sikidom {
public:
  double kerulet() const override; // OK

  /* Forditasi HIBA! Nem definial felul semmit */
  double kerulet() override;

  /* Forditasi HIBA! Sikidom::tipus nem virtualis */
  const string& tipus() const override;
};
```

final metódus

- Ha meg akarjuk akadályozni, hogy a leszármazott osztályokban valamely metódus felül legyen definiálva, akkor azt a `final` azonosítóval tehetjük meg
  - A metódus deklarációja mögött kell megadni
    - Hasonlóan az `override`-hoz
    - Ennek a kettőnek a sorrendje tetszőleges

```
class Sikidom {
public:
  virtual double kerulet() const;
};

class Haromszog : public Sikidom {
public:
  double kerulet() const override final; // OK
};

class Derekszoguharomszog : public Haromszog {
public:
  // HIBA! final metodus van feluldefinilva
  double kerulet() const override;
};
```

final osztály

- Arra is van lehetőség, hogy megtiltsuk, hogy az adott osztályból leszármazzon másik osztály
  - Ilyenkor magát az osztályt kell `final`-lé tenni
    - Az osztály deklaráció mögött kell megadni

```
class Sikidom {
public:
  virtual double kerulet() const;
};

class Haromszog final : public Sikidom {
public:
  double kerulet() const override;
};

// HIBA! az ososztaly final
class Derekszoguharomszog : public Haromszog {
public:
  double kerulet() const override;
};
```

override, final

Nem kulcsszavak

- Az override és a final a C++11 szabványban jelentek meg
- A bővítés célja a nyelv fejlődése
  - A kompatibilitás megtartásával
    - Ha eddig lefordult egy program, akkor ezután is le kell fordulnia
- Ha az új szabványban ezek kulcsszavak lennének
  - Ha valaki használta volna korábban az `override` vagy `final` szavakat például azonosítóként, akkor nem fordulna a programja
    - Elvesztenénk a kompatibilitást
- Ezek nem lettek a C++ kulcsszavai

```
int main() {
  bool override = true; // legalis C++ kod
  // ...
}
```

„Interfész”

```
class Sikidom {
public:
  virtual double kerulet() const { return -1; }
};

class Sokszog : public Sikidom {
};

class Haromszog : public Sokszog {
  double a, b, c;
public:
  Haromszog(double a, double b, double c) : a(a), b(b), c(c) { }
  double kerulet() const override { return a+b+c; }
};

class Negyzet : public Sokszog {
  double a;
public:
  Negyzet(double a) : a(a) { }
  double kerulet() const override { return 4*a; }
};
```

„Interfész”

- Olyan alakzat „nincs”, hogy síkidom
  - Általánosan nem tudom megmondani a kerületét
    - Értelmetlen is lenne meghívni a kerület metódusát
    - Nem is szabadna implementálni
  - Csak interfészként szolgál az osztály
    - Nem is szabadna példányosítani
- Csak a háromszög vagy a négyzet példányosításának van értelme

```
void kiir(const Sikidom &s) {
  cout << "Kerulet = " << s.kerulet() << endl;
}

int main() {
  Haromszog h(3, 4, 5);
  kiir(h);
  Negyzet n(2);
  kiir(n);
}
```

Absztrakt osztály, tisztán virtuális metódus

- Ha csak interfészt definiálunk, akkor lehetnek olyan metódusai, amelyeket nem tudunk vagy nem akarunk megvalósítani
  - Például a `kerulet` metódus a síkidom esetében
- Ekkor a virtuális metódust tisztán virtuálisan kell deklarálni
  - A metódus deklaráció végére oda kell írni, hogy **= 0;**

```
class Sikidom {
public:
  virtual double kerulet() const = 0;
};
```

  - Nem kell megvalósítani a metódust
- Tisztán virtuális metódussal rendelkező osztály absztrakt osztály lesz
  - Nincs rá külön kulcsszó
  - Azért lesz absztrakt, mert van tisztán virtuális metódusa
    - Lehet örökölt is
  - Nem lehet példányosítani az osztályt

Absztrakt osztály

Példa

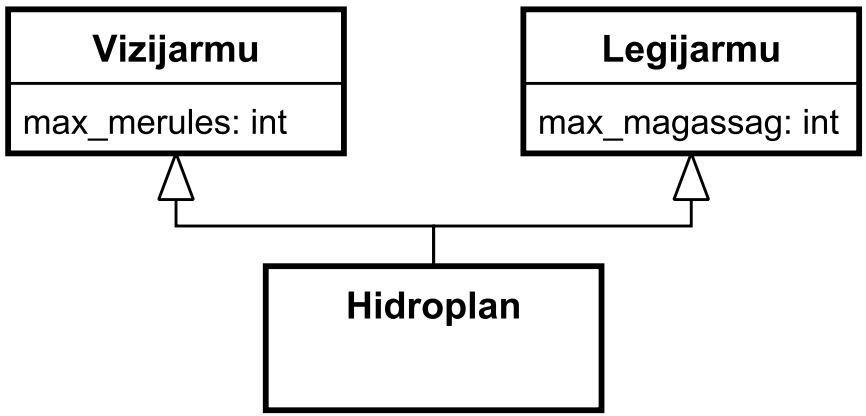
```
class Sikidom {
public:
  virtual double kerulet() const = 0;
};

class Sokszog : public Sikidom {
};

class Haromszog : public Sokszog {
  double a, b, c;
public:
  Haromszog(double a, double b, double c) : a(a), b(b), c(c) { }
  double kerulet() const override { return a+b+c; }
};

int main() {
  Sikidom sikidom; // HIBA! nem peldanyosithato
  Sokszog sokszog; // HIBA! nem peldanyosithato
  Haromszog haromszog(3, 4, 5);
}
```

Tartalom

Többszörös öröklődés

Példa

- Vannak olyan esetek, amikor az osztály kettő vagy több osztály speciális esete
  - Több ősosztályra van szükség
- Például ha van egy `Vizijarmu` és egy `Legijarmu` osztály, és egy `Hidroplan` osztályt akarunk létrehozni
  - Célszerű örököltetni a `Vizijarmu` osztályból, le tud szállni a vízre
  - Ugyanakkor repülni is tud, ezért `Legijarmu` is egyben



Többszörös öröklődés

- C++-ban lehetőség van „valódi” többszörös öröklődés megvalósítására
  - A Java esetében legfeljebb egy osztály lehet a közvetlen ősök között, de tetszőleges számú interfész
- Az ősosztályokat vesszővel elválasztva kell felsorolni
  - Meg kell adni minden egyes osztály esetében az öröklődés láthatóságát
    - Ezek lehetnek különbözőek
    - Az egyes tagokat az ősosztályból az adott láthatóság szerint örökli

```
class Hidroplan : public Vizijarmu, public Legijarmu {
  /* ... */
};
```

- Örökli az ősosztályok tagjait

Többszörös öröklődés

Példa

```
class Vizijarmu {
public:
  const unsigned maxMerules;
  Vizijarmu(unsigned m) : maxMerules(m) {}
};

class Legijarmu {
public:
  const unsigned maxMagassag;
  Legijarmu(unsigned m) : maxMagassag(m) {}
};

class Hidroplan : public Vizijarmu, public Legijarmu {
public:
  Hidroplan(unsigned me, unsigned ma) :
    // konstr. inic. listaban a konstruktorok meg van hivva
    Vizijarmu(me), Legijarmu(ma)
  { }
};
```

Többszörös öröklődés

Példa (folyt.)

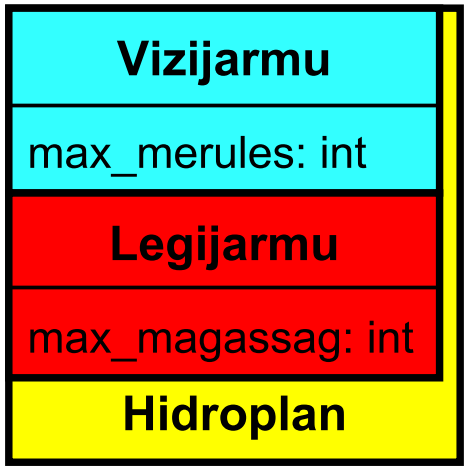
```
void merules(const Vizijarmu& v) {
  cout << "Max. merules: " << v.maxMerules << endl;
}

void magassag(const Legijarmu& l) {
  cout << "Max. magassag: " << l.maxMagassag << endl;
}

int main() {
  Hidroplan hidroplan(1, 2000);
  merules(hidroplan);
  magassag(hidroplan);
}
```

- Kimenet

  ```
  Max. merules:  1
  Max. magassag:  2000
  ```

Örökölt adattagok, inicializálás

- Az osztály örökli az ősosztályok adattagjait
  - „Tartalmazni” fogja mindkét ősosztályt
- Az ősosztályok adattagjainak inicializálása
  - Hasonlóan történik, mint az egyszeres öröklődésnél
  - Meg kell hívni az ősosztályok konstruktorait



- Az objektum olyan sorrendben lesz inicializálva, ahogy az öröklődésnél fel vannak sorolva az ősosztályok
  - A konstruktorok meghívásának sorrendje meg kell egyezzen az öröklődésnél megadott osztályok sorrendjével
  - A saját adattagokat ezek után inicializálhatjuk

```
class Hidroplan : public Vizijarmu, public Legijarmu {
    Hidroplan(unsigned mer, unsigned mag) :
       Vizijarmu(mer),
       Legijarmu(mag)
       /* nincs sajat adattag */
```

Névütközés

```
class Vizijarmu {
public:
  const unsigned maxSebesseg;
  Vizijarmu(unsigned s) : maxSebesseg(s) {}

  virtual void halad() const {
    cout << "A vizijarmu halad: " << maxSebesseg << endl;
  }
};

class Legijarmu {
public:
  const unsigned maxSebesseg;
  Legijarmu(unsigned s) : maxSebesseg(s) {}

  virtual void halad() const {
    cout << "A legijarmu halad: " << maxSebesseg << endl;
  }
};

class Hidroplan : public Vizijarmu, public Legijarmu {
public:
  const unsigned maxSebesseg;
  Hidroplan(unsigned m0, unsigned m1, unsigned m2) :
    Vizijarmu(m1), Legijarmu(m2), maxSebesseg(m0) {
  }

  virtual void halad() const {
    cout << "A hidroplan halad: " << maxSebesseg << endl;
  }
};
```

Névütközés feloldása

- Az ősosztályból örökölt tagokra lehet scope-olt névvel hivatkozni
  - Ha „elrejtjük” az örökölt tagot, akkor csak így tudjuk elérni
  - `Ososztaly::tag`

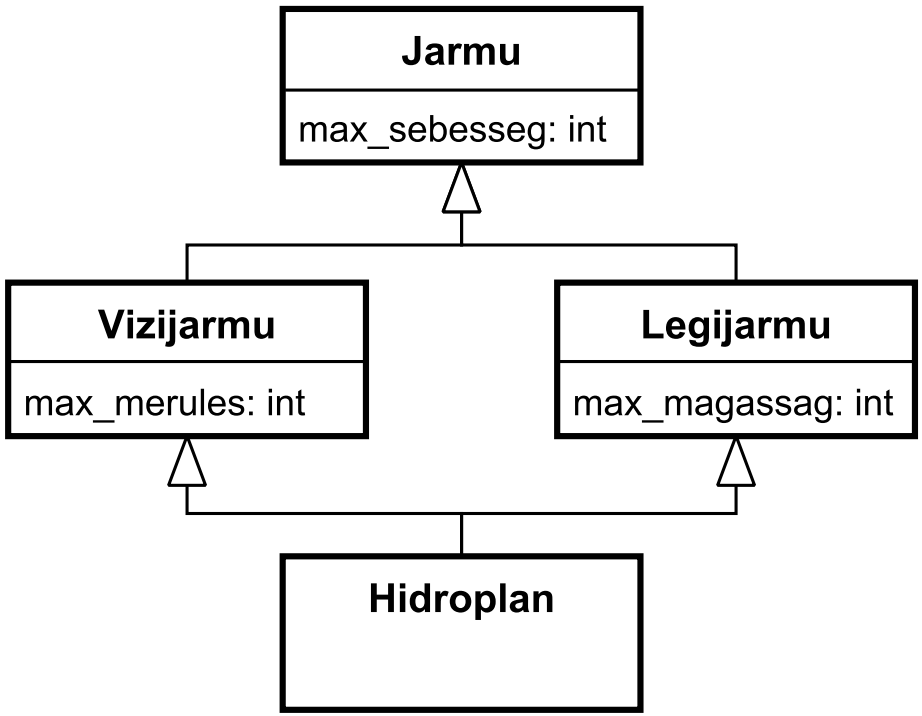
```
int main() {
  Hidroplan hidroplan(100, 40, 300);
  hidroplan.halad();
  hidroplan.Legijarmu::halad();
  cout << hidroplan.Vizijarmu::maxSebesseg << endl;
}
```

- Az osztályon belül is ezt kell használni

```
class Hidroplan : public Vizijarmu, public Legijarmu {
  /* ... */

  virtual void halad() const {
    Vizijarmu::halad();
  }
};
```

Gyémánt öröklődés

- A `Vizijarmu` és `Legijarmu` osztályokban is van közös
  - Mind a kettő jármű, ezért célszerű lenne a „jármű tulajdonságát” egy közös ősosztály segítségével megvalósítani
    - Például max. sebesség tulajdonsága lehetne a járműnek

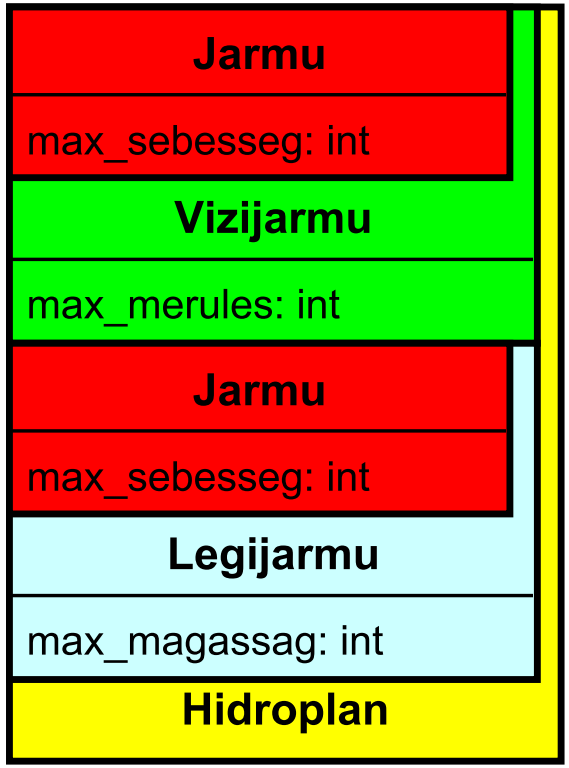


Gyémánt öröklődés (első változat)

```
class Jarmu {
public:
  const unsigned maxSebesseg;
  Jarmu(unsigned vs) : maxSebesseg(vs) { }
};

class Vizijarmu : public Jarmu {
public:
  const unsigned maxMerules;
  Vizijarmu(unsigned s, unsigned m) : Jarmu(s), maxMerules(m) {}
};

class Legijarmu : public Jarmu {
public:
  const unsigned maxMagassag;
  Legijarmu(unsigned s, unsigned m) : Jarmu(s), maxMagassag(m){}
};

class Hidroplan : public Vizijarmu, public Legijarmu {
public:  Hidroplan(unsigned s, unsigned me, unsigned ma) :
           Vizijarmu(s, me), Legijarmu(s, ma) { }
};
```

Gyémánt öröklődés problémája

- A `Jarmu` rendelkezik egy adattaggal
  - `maxSebesseg`
- A `Vizijarmu` a `Jarmu`-ből öröklődik
  - Örökli az adattagját
- A `Legijarmu` is a `Jarmu`-ből öröklődik
  - Itt is lesz egy `maxSebesseg` adattag
- A `Hidroplan` öröklődik a `Vizijarmu` és a `Legijarmu` osztályokból
  - Mind a kettőnek volt egy (örökölt) `maxSebesseg` adattagja
  - Mind a kettőt megörökli a `Hidroplan` osztály



- Csak egy `maxSebesseg` kellene a `Jarmu` osztályban
  - Most még lenne értelme a vízen és a levegőben megkülönböztetni a max. sebességet
    - Például a jármű színe esetében erről már nem beszélhetnénk

Gyémánt öröklődés problémája (folyt.)

- Nem elég azt mondani, hogy sebesség

```
int main() {
  Hidroplan hidroplan(35, 1, 2000);
  cout << hidroplan.maxSebesseg << endl; // HIBA!
}
```

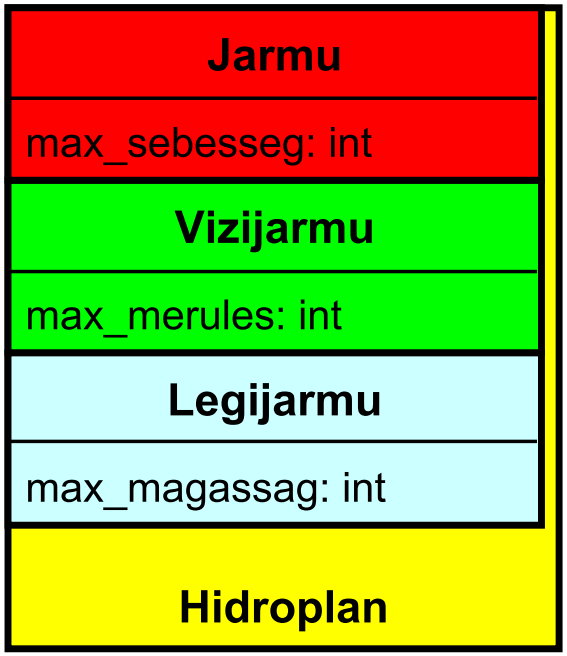
- Mivel kettő `maxSebesseg` van, ezért meg kell mondani, hogy melyikre hivatkozok
  - Melyik ősosztályban lévőre

```
cout << hidroplan.Vizijarmu::maxSebesseg << endl;
cout << hidroplan.Legijarmu::maxSebesseg << endl;
```

- Objektum mérete is „feleslegesen nagyobb”
  - Van benne kettő `maxSebesseg`

Virtuális öröklődés

- A probléma megoldása az lenne, hogy ha a `Jarmu` osztály csak egyszer szerepelne a `Hidroplan` osztályban
- Ehhez virtuálisan kell örököltetni a `Vizijarmu` és `Legijarmu` osztályokat a `Jarmu` osztályból
  - Az öröklődés ezen két osztály esetében nem változik
  - Örökölnek mindent a `Jarmu` osztályból
    - Enélkül nem lennének „teljesek”



- A változás a `Hidroplan` osztály esetében lesz
  - A `Jarmu` osztály csak „egyszer lesz része”
    - Része lesz, mert különben a `Vizijarmu` és `Legijarmu` osztályok nem lennének használhatóak
  - A `Hidroplan` osztálynak kell inicializálnia
    - A két ősosztály nem fogja inicializálni

Virtuális öröklődés

Példa

```
class Jarmu {
public:
  const unsigned maxSebesseg;
  Jarmu(unsigned vs) : maxSebesseg(vs) { }
};

class Vizijarmu : public virtual Jarmu { /* virtual */
public:
  const unsigned maxMerules;
  Vizijarmu(unsigned s, unsigned m) : Jarmu(s), maxMerules(m) {}
};

class Legijarmu : public virtual Jarmu { /* ... */ };

class Hidroplan : public Vizijarmu, public Legijarmu {
public:
  Hidroplan(unsigned s, unsigned me, unsigned ma) :
    Jarmu(s),
    Vizijarmu(s, me), // nem fogja inicializalni a Jarmu reszt
    Legijarmu(s, ma)  // nem fogja inicializalni a Jarmu reszt
  { }
};

int main() {
  Hidroplan hidroplan(35, 1, 2000);
  // nem kell megmondani, hogy melyik maxSebesseg ez, csak egyet orokolt
  cout << hidroplan.maxSebesseg << endl;
}
```

Tartalom

Objektum példányosítása, élettartalma

- Hogyan hozhatunk létre objektumokat?
  - A lenti példa esetében hány objektum jön létre?
  - Milyen sorrendben jönnek létre?
  - Milyen „egyéb módon” lehet még létrehozni?
- Meddig „élnek”?
  - Mikor szűnnek meg?
  - Automatikusan megszűnnek?
  - Milyen sorrendben szűnnek meg?

```
class Macska {
  string nev;
public:
  Macska(const string& nev) : nev(nev) { }
};

int main() {
  Macska cirmos("Cirmos"), foltos("Foltos");
}
```

Speciális tagfüggvények

Konstruktor, értékadó operátor, destruktor

- Konstruktor: objektum létrejöttekor automatikusan lefut
  - Default konstruktor: paraméter nélküli konstruktor
  - Másoló konstruktor: paramétere az adott típus
    - Érték, referencia, konstans referencia
- Értékadó operátor (`operator=`): paramétere az adott típus
  - Érték, referencia, konstans referencia
- Destruktor: az objektum megszűnésekor hívódik automatikusan
  - Neve: ~ és az osztály neve
  - Paramétere nem lehet

Konstruktor, értékadó operátor, destruktor - demo

Példa

```
class Macska {
  string nev = "Cirmi";
public:
  // default
  Macska() {
    cout << "Default: " << nev << endl;
  }

  // parameteres
  Macska(const string& n) : nev(n) {
    cout << "Parameteres: " << nev << endl;
  }

  // masolo
  Macska(const Macska& m) : nev(m.nev) {
    cout << "Masolo: " << nev << endl;
  }

  // ertekado operator
  Macska& operator=(const Macska& m) {
    cout << "Ertekado: " << nev
         << " -> " << m.nev << endl;
    nev = m.nev;
    return *this;
  }

  ~Macska() { // destruktor
    cout << "Destruktor: " << nev << endl;
  }
};
```

```
void macskaTeszt(Macska m) {
  cout << "macskaTeszt" << endl;
}

int main() {
  Macska cirmi;
  Macska foltos("Foltos");
  Macska macska = foltos;
  macska = cirmi;

  cout << "macskaTeszt elott" << endl;
  macskaTeszt(macska);
  cout << "macskaTeszt utan" << endl;
}
```

Objektum létrehozása

Memóriafoglalás

- Amikor egy osztályt példányosítunk, akkor létrejön egy objektum
  - Első lépés a memória lefoglalása
- Lefoglalódik annyi memória, amennyi az objektum tárolásához szükséges
  - A memóriafoglalás automatikusan történik
    - Kivétel az `operator new (size_t size, void* ptr)`
- A `sizeof` operátor megadja az objektum méretét
  - Az összeg nem feltétlen a részek összege
  - A „dinamikusan” foglalt rész nem tartozik bele

```
class C1 {
  bool b1;
  int i;
  bool b2;
};
// sizeof(C1)=12
```

```
class C2 {
  bool b1;
  bool b2;
  int i;
};
// sizeof(C1)=8
```

```
class S3 {
  int *p =
    new int[99];
  int t[5];
};
// 8 + 5*4 = 28
```

Objektum létrehozása

Inicializálás

- A memória lefoglalása után megtörténik az objektum inicializálása
- A megfelelő konstruktor meghívódik
  - Ha nem adunk meg paraméter, akkor a default konstruktor fog meghívódni
  - Az objektum adattagjai is inicializálódnak
    - Kezdőérték beállítása
    - Objektumok inicializálása
  - A konstruktor törzse lefut

```
class Macska {
  unsigned ev; // 1. erteket kap
  string nev;  // 2. meghivodik a megfelelo konstruktor
public:
  Macska(unsigned ev, const string& nev) : ev(ev), nev(nev){
    // 3. vegrehajtodik a konstruktor torzse
  }
};
```

Objektum megszűnése

- Ugyanazok a lépések játszódnak le, csak „visszafele”
- Destruktor lefutása
  - Destruktor törzse lefut
- Objektum megszűnése
  - Adattagok megszűnése
    - Létrehozással ellentétes sorrendben
    - Meghívódik azoknak is a destruktora
- Memória felszabadul

```
class Macska {
  int ev;       // 3. ha lenne, meghivodna a destruktora
  string nev;   // 2. meghivodik a destruktor
public:
  Macska(int ev, const string& nev) : ev(ev), nev(nev) { }
  ~Macska() {
    // 1. lefut a torzse
  }
};
```

Hol jöhetnek létre objektumok?

Hol jöhetnek létre objektumok?

- Verem (stack)
  - Függvények/metódusok lokális változói
- Kupac (heap)
  - Dinamikusan foglalt objektumok
- Statikus
  - Metódusban
  - Osztályban
- Globális
  - Nem függvény/metódus/osztály része
  - Globális névtér
    - Vagy valamelyik névtérben is lehet

Tartalom

Pointer - demo

- Pointer: olyan változó, amely nem „közvetlenül tárolja az információt”, hanem egy „másik memória területre” mutat
  - Ha nem adunk értéket a változónak, akkor nem definiált hova mutat
  - Értéket adhatunk például
    - Dinamikus memória foglalással
    - Másik változó címét adjuk meg
    - `nullptr`
- Ha egy másik változó címét adjuk meg, akkor annak a memória területnek „2 neve van”
  - Változó memóriacímét a & operátorral tudjuk elkérni
  - Mindegy, hogy az `i`-t vagy a `*p`-t használjuk

```
int i = 5;
int *p = &i;
cout << "i = " << i << endl;   // i = 5
++*p;
cout << "i = " << i << endl;   // i = 6
++i;
cout << "*p = " << *p << endl; // *p = 7
```

Referencia és pointer összehasonlítása

- Deklaráció
  - A referenciának deklarációnál értéket kell adni
  - A pointernek nem szükséges
    - A pointer lehet definiálatlan, vagy lehet `nullptr`
- „Alias másik változóra”
  - A pointernek új értéket is adhatunk

```
int i = 5, j = 6;
int *p = &i; // p az i-re mutat
p = &j;      // p a j-re mutat
```

  - A referenciát nem lehet megváltoztatni
- Pointeraritmetika
  - Pointer esetében van pointeraritmetika `++p;`
  - Referencia esetében nincs
- Használat
  - Pointer: `*p = 3;` `(*p)++;` `p->i = 5;`
  - Referencia: `r = 3;` `r++;` `r.i = 5;`

Dinamikusan létrehozott objektumok

- Vannak esetek, mikor hosszabb élettartalmú objektumot használnánk
  - A main függvényben létrehozott objektum élettartalma elég hosszú ☺
    - De ez meg már túl hosszú
- A `new` operátorral (vagy a `malloc` segítségével) létrehozott objektumok a kupacon (heap) jönnek létre
  - Másik memóriaterület, mint a verem
- Egy pointert kapunk az objektumra
  - Lefoglalja a szükséges memóriát
  - Inicializálhatja az adott memóriaterületet
    - A `new` operator meghívja a megfelelő konstruktort
    - calloc is inicializál
  - Visszaadja a létrejött objektumra mutató pointert
- Fontos: **mi felelünk a memória felszabadításáért**
- Ha elmarad a felszabadítás, akkor
  - Nem fog lefutni a destruktor
    - Az objektum nem lesz megfelelően „lezárva”
  - A program végéig feleslegesen foglaljuk a memóriát
    - Elfogyhat a memória

C++ dinamikus memóriafoglalás

Példa egyszerű típusokra

- Memória foglalás: **new** és **new []** operátorok segítségével történik
- Memória felszabadítás: **delete** és **delete []** operátorok

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

int main() {
  int *i = new int;     // 1 db int
  int *p = new int[3]; // 3 elemu int tomb
  *i = 6;
  *p = *i + 4;
  *(p+1) = 9;
  p[2] = 21;
  cout<< *i << " " << *p << " " << p[1] << " " << *(p+2) << endl;
  delete i;   // new parja
  delete[] p; // new[] parja
}
```

- Kimenet

```
6 10 9 21
```

Dinamikus memóriafoglalás

Példa objektumokra

```
class Macska {
  string nev;
  int szulEv;
public:
  Macska(const string& nev, int szEv) :
    nev(nev), szulEv(szEv) { /* ures */ }
  void jatszik() { /* ... */ }
};

Macska* macskaBeolvas() {
  string nev; int ev;
  cin >> nev >> ev;
  Macska *k = new Macska(nev, ev); // letrejon a Macska tip. obj.
  return k;
} // nem szunik meg az objektum

int main() {
  Macska *macska = macskaBeolvas();
  macska->jatszik();  // hasznalom ...
  delete macska;      // nekem kell torolni
}
```

Objektumok inicializálása

- Osztályok esetében a megfelelő konstruktorral inicializálva lesz
  - Nem tudjuk inicializálatlanul létrehozni
    - Ha nincs default konstruktor, akkor „nem tudunk tömböt foglalni”

```
class Macska {
  string nev = "Cirmi";
public:
  Macska() {
    cout << "Macska(): " << nev << endl;
  }
  Macska(const string& nev) : nev(nev) {
    cout << "Macska(const string&): " << nev << endl;
  }
};

int main() {
  Macska *m1 = new Macska;           // def. konstr., Macska(): Cirmi
  Macska *m2 = new Macska("Foltos");// par. konstr., Macska(const string&): Foltos
  Macska *t  = new Macska[10];       // def. konstr., Macska(): Cirmi, ...

  // par. konstr. fog hivodni ketszer
  Macska *k = new Macska[2] { Macska("Cirmi"), "Foltos" };

  // par. konstr. fog hivodni ketszer, majd a def. egyszer
  Macska *h = new Macska[3] { Macska("Cirmi"), "Foltos" };

  // torlesek ...
}
```

Beépített típusok inicializálása

- Beépített típusok esetében a dinamikus foglalás esetén nem inicializálódnak a változók

```
int main() {
  int *i = new int;         // nem definialt a *i erteke
  int *t = new int[1000]; // nem def. a tomb elemeinek erteke
}
```

- Létrehozáskor itt is lehetőségünk van inicializálni vagy kinullázni

```
int main() {
  int *i = new int(3);         // *i erteke 3
  int *t1 = new int[1000](); // minden elem erteke 0
  int *t2 = new int[1000]{}; // minden elem erteke 0

  // az elso 5 elem erteke 5, 4, 3, 2, 1
  // minden tovabbi elem erteke 0
  int *t3 = new int[1000]{5, 4, 3, 2, 1};
}
```

new operátor

```
void* operator new (std::size_t size);
```

- `size` bájtot allokál
- Egy nem-null pointerrel tér vissza, amelyik az első bájtra mutat
- Hiba esetén `bad_alloc` kivételt dob

```
void* operator new (std::size_t size, const std::nothrow_t& nt) noexcept;
```

- Ugyanaz, mint az előző, csak hiba esetén kivétel dobása helyett null pointerrel tér vissza
- A második paramétere a konstans `nothrow`
  - Csak azért kell, hogy megkülönböztessük a másik `operator new`-tól

```
void* operator new (std::size_t size, void* ptr) noexcept;
```

- **Nem foglal memóriát**
  - A `ptr` által mutatott címre történik az inicializálás
- A `ptr` által mutatott címmel tér vissza

Különböző new operátorok

Példa (felszabadítás nélkül)

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

int main() {
  int *i1 = new int;             // throwing allocation
  int *i2 = new (nothrow) int; // nothrow allocation
  int *i3 = new (i2) int;        // placement allocation
  cin >> *i1 >> *i2 >> *i3;
  cout << *i1 << " " << *i2 << " " << *i3 << endl;
  cout << i1 << " " << i2 << " " << i3 << endl;
}
```

- Példa bemenet: 1 2 3
- Kimenet (`i2` és `i3` memória címe ugyanaz)

```
1 3 3
0x1090c20 0x1090c40 0x1090c40
```

C++ dinamikus memóriafoglalás ellenőrzése

- A memória foglalása csak akkor sikeres, ha van elég memória
  - A sikerességet ellenőrizni kell
- Hiba esetén a `new (std::size_t size);` kivételt dob
  - `std::bad_alloc` kivételt kell elkapni
- Példa
  - Sikertelen memória foglalás esetén hibaüzenetet fog kiírni és kilép

```
int main() {
  const int LIMIT = 10000000000;

  int *ip;
  try {
    ip = new int[LIMIT];
  } catch (const bad_alloc& error) {
    cout << "Nem sikerult a memoria foglalas" << endl;
    return 1;
  }
  cin >> ip[0] >> ip[LIMIT-1];
  cout << ip[0] << " " << ip[LIMIT-1] << endl;
}
```

C++ dinamikus memóriafoglalás ellenőrzése

- A memória foglalása csak akkor sikeres, ha van elég memória
  - A sikerességet ellenőrizni kell
- Hiba esetén a `new (nothrow);` null pointerrel tér vissza
  - Nem dob kivételt
- Példa
  - Sikertelen memória foglalás esetén hibaüzenetet fog kiírni és kilép

```
int main() {
  const int LIMIT = 10000000000;

  int *ip = new (nothrow) int[LIMIT];
  if (!ip) {
    cout << "Nem sikerult a memoria foglalas" << endl;
    return 1;
  }
  cin >> ip[0] >> ip[LIMIT-1];
  cout << ip[0] << " " << ip[LIMIT-1] << endl;
}
```

C++ dinamikus memória foglalás és felszabadítás

operator delete

```
void operator delete (void* ptr) noexcept;
```

- Felszabadítja a memóriát, amire a `ptr` mutat
  - `new`-val történő foglalást lehet csak felszabadítani
  - Null pointer esetén nem csinál semmit sem (nem okoz hibát)
- *double free or corruption* hibát kapunk, ha már felszabadítottuk
  - Célszerű null pointerre állítani a pointert felszabadítás után

```
delete p;
p = nullptr;
```

```
void operator delete (void* ptr, const std::nothrow_t&) noexcept;
```

- Ugyanaz, mint az előző
  - Az alap implementáció az előzőt hívja

```
void operator delete (void* ptr, void* voidptr2) noexcept;
```

- Semmit sem csinált
- A `voidptr2` nincs használva az alap implementációban

C++ dinamikus memória foglalás és felszabadítás

Memória felszabadítás példa

```
#include <iostream>

using namespace std;

int main() {
  int *i1 = new int;              // throwing allocation
  int *i2 = new (nothrow) int;    // nothrow allocation

  cin >> *i1 >> *i2;
  cout << *i1 << " " << *i2 << endl;

  delete i1;
  i1 = nullptr;
  delete i2;
  i2 = nullptr;
}
```

C++ dinamikus memória foglalás és felszabadítás

operator new[] és operator delete[]

- Hasonló a **`new`** és **`delete`** operátorokhoz
  - Ugyanazok a kiterjesztések vannak
  - Ugyanúgy memóriát foglalnak vagy szabadítanak fel
  - Ugyanazok a hibakezelések vannak
- Különbség a `malloc`-hoz képest: inicializálja az objektumokat
  - Meghívja a konstruktorokat

```
int main() {
  int *t = new int[2];
  cin >> t[0] >> t[1];
  cout << t[0] << "," << t[1] << endl;
  delete[] t; // nem szabad a delete-et hasznalni
}
```

- **`new`** [] -val allokált memóriát **`delete`** [] -tel kell felszabadítani
  - Ha a **`delete`** operátort használjuk a **`delete`** [] helyett vagy fordítva, akkor **nem definiált viselkedés** lesz az eredménye

Objektum másolása - demo, cppinsights

Mi hívódik?

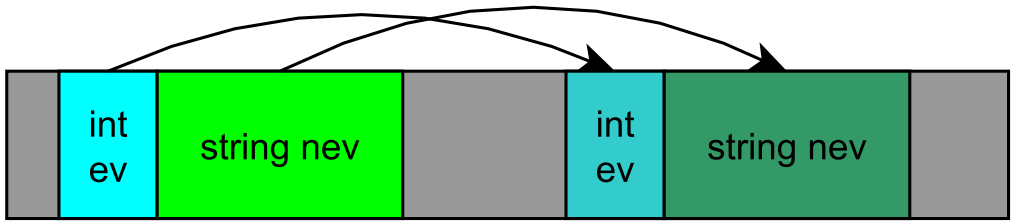
- Eddig nem foglalkoztunk az objektumok másolásával
  - A fordítóprogram megoldotta helyettünk
    - Vajon mindig meg tudja oldani?

```
class Macska {
  unsigned ev;
  string nev;
public:
  Macska(unsigned ev, const string& nev = "") : ev(ev), nev(nev) { }

  void setNev(const string& ujNev) {
    nev = ujNev;
  }

  friend ostream& operator<<(ostream& os, const Macska& m) {
    return os << m.nev << " (" << m.ev << ")";
  }
};

int main() {
  Macska *foltos = new Macska(2023, "Foltos");
  Macska cirmi(*foltos);
  cirmi.setNev("Cirmi");

  cout << *foltos << endl;
  delete foltos;
  cout << cirmi << endl;
}
```

„Automatikusan másolható” objektumok - demo

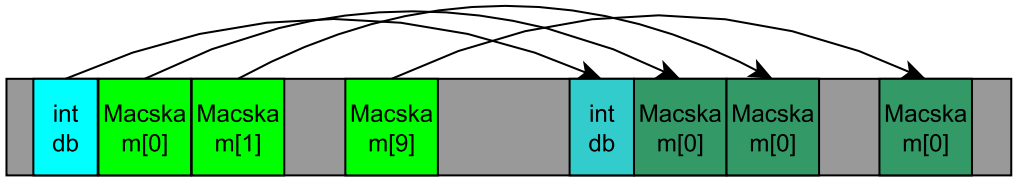
- A fordító által készített „másolás” az objektumot másolja le
  - Az adott memóriaterületet másolja
    - Az ott tárolt adatok lesznek lemásolva
  - Két adattag (a `string` az „másolható”)

```
class Macska {
  int ev; // unsig.
  string nev;
};
```



  - Nem dinamikusan foglalt tömb (Macska „másolható”)

```
class Menhely {
  int db; // unsig.
  Macskak m[10];
};
```



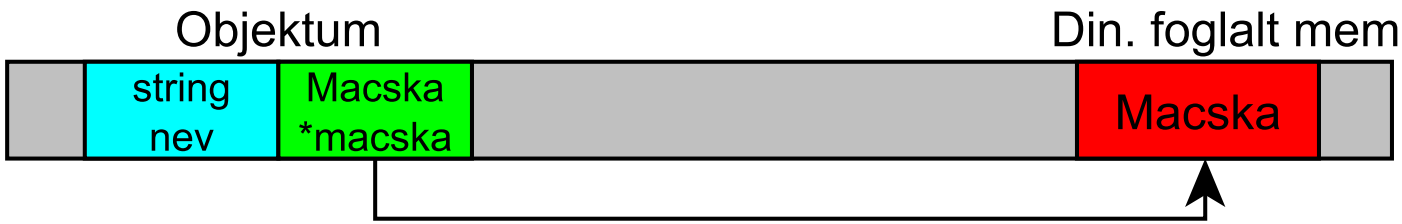
  - Vector (tároló) esetén is működik (Macska „másolható”)

```
class Menhely {
  vector<Macska> macskak;
};
```

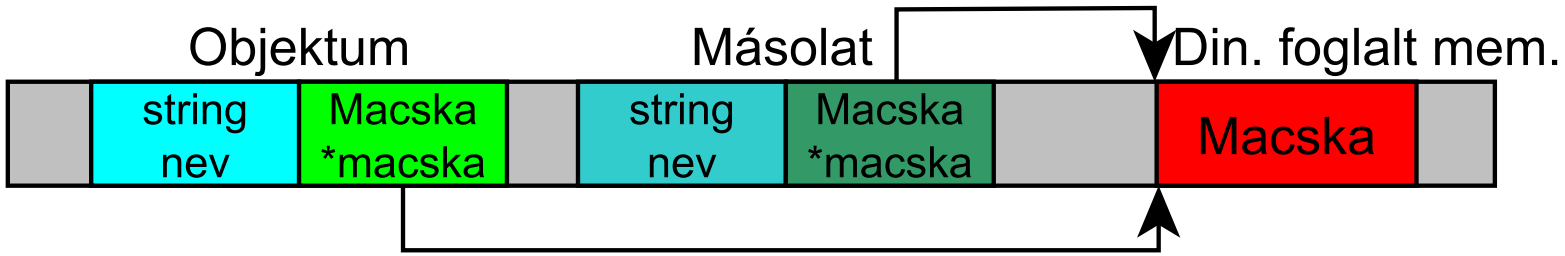
Objektum másolása

Amikor az „objektum bitenkénti másolása” nem elég

- A fordító által készített „másolás” az objektumot másolja le
  - Nem ismeri az osztály logikáját
    - Nem tudja „megfelelően” lemásolni



- A `Gazda` példa esetében nem elég lemásolni az adattagokat, mert akkor a dinamikusan foglalt memória terület közös lesz

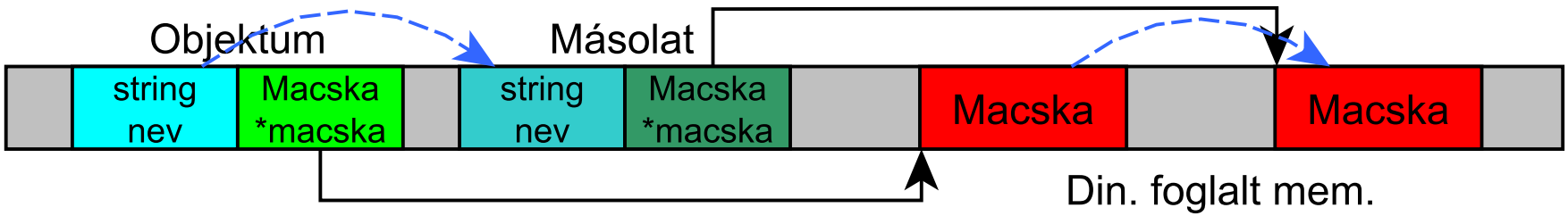


  - Ha az egyik macskát módosítjuk, akkor a másik is változik
  - Ha az egyik `Macska` objektum megszűnik, akkor felszabadítja az általa foglalt memóriát
    - A másik `Macska` érvénytelen területen dolgozik
    - A másik is megpróbálja felszabadítani, és akkor hibával leáll a program

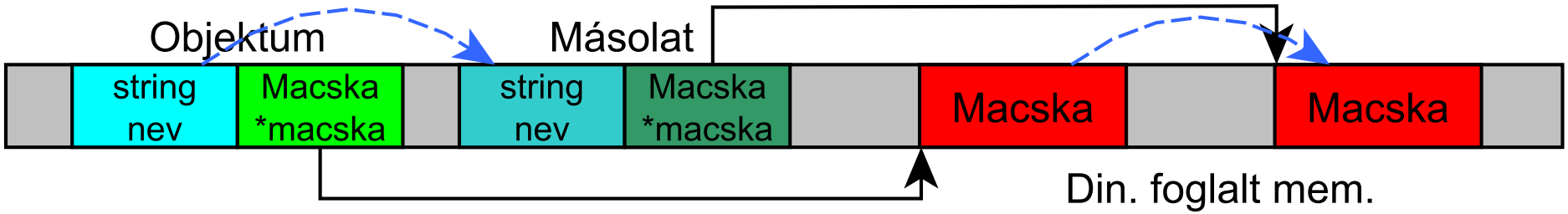
Objektum másolása

Saját másolás definiálása

- Ha az objektum „külső erőforrást” is használ, amit az objektum allokált, akkor a másolatnak is biztosítani kell ezt az erőforrást
  - A `Gazda` esetében minden objektumnak saját dinamikusan foglalt memóriaterülettel kell rendelkeznie



Objektum másolása



- Mire figyeljünk másolásnál?
  - Nekünk kell lemásolni magát az objektumot is!
    - A nem dinamikusan foglalt részt is
  - Erőforrást (pl. memóriát) le kell foglalni
    - Ezt is át kell másolni
  - Értékadó operátor
    - Ha az objektum már foglalt memóriát, azt fel kell szabadítani
    - Önmagának történő értékadás ellenőrzése
- „Ki másoljon”?
  - Értékadó operátor (assingment operator)
    - `T& operator=(const T& o)`
  - Másoló konstruktor (copy constructor)
    - `T(const T& o)`

Másoló konstruktor, értékadó operátor

```
class Gazda {
  string nev;
  Macska *macska = nullptr;
public:
  Gazda(const string& nev) : nev(nev) { }

  Gazda(const Gazda& gazda) :              // masolo konstruktor
    nev(gazda.nev),
    macska(gazda.macska ? new Macska(*gazda.macska) : nullptr)
  { }

  Gazda& operator=(const Gazda& gazda) { // ertekado operator
    if (this == &gazda)
      return *this;

    delete macska;
    macska = gazda.macska ? new Macska(*gazda.macska) : nullptr;
    nev = gazda.nev;

    return *this;
  }
};
```

Ki fogja felszabadítani a memóriát?

- A destruktorban nekünk kell megvalósítani

```
class Gazda {
  string nev;
  Macska *macska = nullptr;
public:
  Gazda(const string& nev) : nev(nev) { }

  Gazda(const Gazda& gazda) :              // masolo konstruktor
    nev(gazda.nev),
    macska(gazda.macska ? new Macska(*gazda.macska) : nullptr)
  { }

  Gazda& operator=(const Gazda& gazda) { // ertekado operator
    if (this == &gazda)
      return *this;

    delete macska;
    macska = gazda.macska ? new Macska(*gazda.macska) : nullptr;
    nev = gazda.nev;

    return *this;
  }

  ~Gazda() {                                 // destruktor
    delete macska;
  }
};
```

Dinamikus adattagok

- Ha az osztály „tartalmaz” dinamikusan foglalt memóriaterületet
  - A mi felelősségünk az objektum megfelelő másolása
    - Másoló konstruktor
    - Értékadó operátor
  - A dinamikusan foglalt memória felszabadítása
    - Destruktor
- A pointer nem jelent automatikusan felszabadítást vagy másolást

```
class Kutya {
  Gazda *gazda = nullptr;
public:
  void setGazda(Gazda* g) {
    gazda = g;
  }
};

int main() {
  Gazda gazda;
  Kutya kutya;
  kutya.setGazda(&gazda);
}
```

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`
  - Osztályok, másolás, „hosszú élet”
  - **Polimorfizmus hiányzik**

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`
  - Osztályok, másolás, „hosszú élet”
  - **Polimorfizmus hiányzik**
- Tárolók és pointer: `vector<Allat*>`

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`
  - Osztályok, másolás, „hosszú élet”
  - **Polimorfizmus hiányzik**
- Tárolók és pointer: `vector<Allat*>`
  - Osztályok, „hosszú élet”, **öröklődés**, **polimorfizmus**

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`
  - Osztályok, másolás, „hosszú élet”
  - **Polimorfizmus hiányzik**
- Tárolók és pointer: `vector<Allat*>`
  - Osztályok, „hosszú élet”, **öröklődés**, **polimorfizmus**
  - **Másolás - hogyan?**

„Ott vagyunk már?”

- Mit tanultunk eddig?
  - Osztályok, öröklődés, polimorfizmus
  - Objektumok másolása
  - Tárolók
  - „Hosszú élet”
    - Dinamikus memória, saját másolás, felszabadítás
- Tárolókba érték szerinti eltárolás: `vector<Allat>`
  - Osztályok, másolás, „hosszú élet”
  - **Polimorfizmus hiányzik**
- Tárolók és pointer: `vector<Allat*>`
  - Osztályok, „hosszú élet”, **öröklődés**, **polimorfizmus**
  - **Másolás - hogyan?**
  - **set<Allat*>**

Tároló és pointer

Hogyan másoljam?

```
class Kerdes { /* ... */ };

class FeleletValasztos : public Kerdes { /* ... */ };

class Kifejtos : public Kerdes { /* ... */ };

class KerdesBank {
  string nev;
  vector<Kerdes*> kerdesek;
public:
  KerdesBank(const string& nev) : nev(nev) { }

  ~KerdesBank() {
    for (auto k : kerdesek)
      delete k;
  }

  KerdesBank& operator+=(Kerdes* kerdes) {
    kerdesek.push_back(kerdes);
    return *this;
  }

  void kiir() const {
    cout << nev << endl;
    for (auto k : kerdesek)
      k->megjelenit();
  }
};
```

Ki definiálja a másolást?

Kérdésbank „intézze a másolást”

- Tudnia kell, hogy mi a `Kerdes` dinamikus típusa
  - Nem operatornak ezt kell megadni
- Megoldások a típus kitalálására
  - Valami enum bevezetése
  - `dynamic_cast` használata
  - Nagy elágazás
    - Több száz típus
- Mi van, ha bevezetnek egy új kérdéstípust?
  - Bővíteni kell mindenhol, ahol másolunk
- Logikailag sem a `KerdesBank` feladata
  - Több helyen is használhatjuk

Klónozható objektumok

- Minden osztály mondja meg magáról, hogy hogyan kell lemásolni
  - Definiálja felül az ősből örökölt másolást
  - Nem azért kell ezt definiálni, mert van dinamikus adattagjuk
    - Hanem azért, hogy lehessen a másolást polimorfikusan használni

```
class Kerdes {
  /* ... */
public:
  virtual ~Kerdes() = default;

  virtual Kerdes* clone() const = 0;
};

class FeleletValasztos : public Kerdes {
  /* ... */
public:
  FeleletValasztos* clone() const override {
    return new FeleletValasztos(*this);
  }
};

class Kifejtos : public Kerdes {
  /* ... */
public:
  Kifejtos* clone() const override {
    return new Kifejtos(*this);
  }
};
```

Polimorfikus másolás

```
class KerdesBank {
  string nev;
  vector<Kerdes*> kerdesek;
public:
  KerdesBank(const string& nev) : nev(nev) { }

  KerdesBank(const KerdesBank& kb) : nev(kb.nev) {
    for (Kerdes* k : kb.kerdesek)
      kerdesek.push_back(k->clone());
  }

  KerdesBank operator=(const KerdesBank& kb) {
    if (this == &kb)
      return *this;

    nev = kb.nev;

    for (Kerdes* k : kerdesek)
      delete k;
    kerdesek.clear();
    for (Kerdes* k : kb.kerdesek)
      kerdesek.push_back(k->clone());

    return *this;
  }

  ~KerdesBank() {
    for (auto k : kerdesek)
      delete k;
  }
};
```

Pointerek és rendezés

- A set-nek szüksége van rendezésre
  - Hogyan tehetek bele pointereket?

```
int main() {
  set<Kerdes*> s;
  s.insert(new Kifejtos(5, "Mondjon ..."));
  s.insert(new Kifejtos(5, "Irjon ..."));

  for (Kerdes* k : s)
    k->megjelenit();

  Kifejtos* k = new Kifejtos(8, "Soroljon fel ...");
  if (!s.insert(k).second)
    cout << "1. proba: \"" << k->getKerdes() << "\" nem lett beszurva." << endl;
  if (!s.insert(k).second)
    cout << "2. proba: \"" << k->getKerdes() << "\" nem lett beszurva." << endl;

  for (Kerdes* k : s)
    k->megjelenit();
}
```

- Lefordul, működik, de **egyedi objektumokat** tárol
  - Nem tehetem bele kétszer ugyanazt az objektumot
    - Ha lemásolom, a másolatot már bele tudom rakni

Halmaz saját rendezéssel

- Megadhatjuk az összehasonlítást a set-nek
  - Definiáljuk a két elem (pointer) közötti a relációt

```
class Kerdes {
  unsigned pont;
public:
  Kerdes(unsigned pontszam = 0) : pont(pontszam) { }
  virtual ~Kerdes() = default;
  unsigned getPont() const noexcept { return pont; }
  void setPont(unsigned ujPontszam) noexcept { pont = ujPontszam; }
  virtual void megjelenit() const = 0;
  virtual Kerdes* clone() const = 0;

  class Osszehasonlitas {
  public:
    bool operator()(const Kerdes* k1, const Kerdes* k2) const {
      return k1->getPont() < k2->getPont();
    }
  };
};

int main() {
  // megadjuk, hogy melyik osztalyt hasznalja osszehasonlitasra
  set<Kerdes*, Kerdes::Osszehasonlitas> s;
  s.insert(new Kifejtos(5, "Mondjon ..."));
  s.insert(new Kifejtos(5, "Irjon ..."));  // nem fogja beszurni, mert pontszam

  for (Kerdes* k : s)
    k->megjelenit();
}
```

Tanácsok a ZH-hoz

- Készülés
  - Házi feladatok megoldása
  - Előadás jegyzet átolvasása
  - „Típushibák” megismerése
  - Segítség kérése
    - discord
- Feladat megoldása
  - Feladat végig olvasása
  - Átgondolni, hogy mit is szeretnék
  - Elkészült részek kipróbálása, tesztelése
- Lokális teszt használata
  - Hibaüzenetek értelmezése
  - Lokális teszt futtatása
    - Ha ez sem megy, akkor a „biro sem ad rá pontot”
  - valgrind használata
  - Saját további tesztek írása

Tartalom

Verem (stack) - demo (stack 1)

- A függvények kódja nem szerepel annyiszor a memóriában, ahányszor meghívódtak
  - De a lokális változóiknak szükséges valahol memóriát foglalni
- Ha egy függvény meghívódik, akkor a veremben tárolódnak
  - A **lokális változói**
  - A paraméterek
  - Egyéb, a működéshez szükséges információ
- „Automatikusan” működik
  - Automatikusan foglalódik le a memória
  - A blokk végén felszabadul, nem nekünk kell gondoskodni róla

```
unsigned lnko(unsigned a, unsigned b) {
  cout << &a << " " << a << " " << b << " " << endl;
  if (a < b)
    return lnko(b, a);
  if (b == 0)
    return 0;
  return lnko(b, a%b);
}
```

Lokális objektumok - demo (stack 2)

- Ha egy függvény (metódus, operátor, ...) valamelyik blokkján belül hozunk létre objektumot, akkor a létrehozástól a blokk végéig fog élni
  - Dinamikus (`new`) és statikusan (`static`) különbözik

```
int main() {
  Macska cirmi;                       // Default: Cirmi

  cout << endl << "blokk elott" << endl;
  {
    Macska szurke("Szurke");          // Parameteres: Szurke
  }                                   // Destruktor: Szurke
  cout << "blokk utan" << endl << endl;

  Macska foltos("Foltos");            // Parameteres: Foltos
}                                     // Destruktor: Foltos
                                      // Destruktor: Cirmi
```

Lokális objektumok - demo (stack 3)

for példa

- Ciklus esetén minden iterációnál új objektum jön létre majd megszűnik

```
int main() {
  string nevek[] = {"Garfield", "Sziamiau"};
  for (const auto& nev : nevek) {
    Macska macska(nev);
    cout << macska << endl;
  }
}
```

- Kimenet

```
Parameteres:  Garfield
Garfield
Destruktor:  Garfield
Parameteres:  Sziamiau
Sziamiau
Destruktor:  Sziamiau
```

Objektumok inicializálása a vermen - demo (stack 4)

Osztályok (class) esetében

- Az objektumok inicializálva lesznek
  - A megfelelő konstruktor automatikusan meghívódik

```
int main() {
  Macska macska;
  Macska foltos("Foltos");
  vector<Macska> ures;
  vector<Macska> otMacska(5);
  vector<Macska> macskak { {"Garfield"}, {"Sziamiau"} };
}
```

További változók inicializálása a vermen

Primitív típusok, pointerek, tömbök

- A primitív típusok nem kapnak kezdőértéket
  - Ha nem adjuk meg, akkor definiálatlan lesz az értékük

```
int main() {
  int i;      // nem definialt az i erteke
  int *p;     // nem definialt, hogy a pointer hova mutat
  int t[10]; // nem definialt a tomb elemeinek erteke
}
```

- A mi feladatunk ezek inicializálása a felhasználás előtt
  - Lehetőségünk van létrehozáskor is inicializálni ezeket
    - Később is adhatunk értéket (létrehozáskor még nem tudjuk inicializálni)

```
int a = 11;
int b(11);
int c{11};
int d = int();  // d = 0;
int *p = nullptr;
int t1[1000] = {};                  // 0, 0, 0, ..., 0
int t2[1000] = {5, 4, 3, 2, 1}; // 5, 4, 3, 2, 1, 0, ..., 0
```

Verem

Előnyök, hátrányok

- Előnyök
  - Automatikus a memóriafoglalás és felszabadítás
  - Gyors
    - Gyorsabb, mint a dinamikus
- Hátrányok
  - Korlátozott a mérete
    - Nem a memória max. mérete a korlát
    - Program indulásánál beállítódik
  - „Korlátozott” az objektumok élettartalma
    - Metódusban létrehozott objektum megszűnik a visszatérés után

static jelentése

- Mi lehet `static`, és mit jelent?

static jelentése

- Mi lehet `static`, és mit jelent?
  - Globális objektum
  - Statikus objektum függvényben vagy metódusban
  - Statikus osztály adattag

Statikus objektum függvényben/metódusban

- Függvényen belül található statikus objektum
  - Statikus globális scope-ban található objektum mást jelent
- A blokk elhagyásakor sem szűnik meg az értéke
  - Második hívásnál is ugyanazt az objektumot használom
  - Nem úgy viselkedik, mint a lokális változók
- Az „első használatkor” jön létre
  - Ugyanúgy, ahogy a lokális változók
  - Automatikusan történik a memória foglalás és inicializálás
    - Nem nekünk kell, mint ahogy a `new` esetében
- A program futásának végén szűnik meg
  - Meghívódik a destruktor
- A static memória területen jön létre
  - A veremtől és a kupactól eltérő memóriaterület

Statikus objektum - demo (static 1)

```
class Counter {
  size_t n = 0;
public:
  Counter& operator++() {
    ++n;
    return *this;
  }
  ~Counter() {
    cout << "Counter: "
         << n << endl;
  }
};
```

```
class Macska {
  string nev = "Cirmi";
public:
  Macska() = default;

  Macska(const string& nev)
   : nev(nev) { }

  void jatszik() const {
    static Counter c;
    ++c;
    // a macska jatszik ...
  }
};

int main() {
  Macska c("Cirmi");
  Macska f("Foltos");
  c.jatszik();
  f.jatszik();
  f.jatszik();
}
```

Statikus tagok

- Minden objektum egyedi
  - De lehetnek olyan tulajdonságok vagy műveletek, amelyek nem az egyedi objektumra érvényesek, hanem magára a típusra
    - Minden objektumra egyformán érvényesek
- Statikus tagok
  - `static` modósítóval definiálhatjuk
- Statikus adattagok
  - A statikus memória területen tárolódik
  - Csak egy darab van belőle
  - Az objektumok osztoznak rajta
    - Példányoítás nélkül is elérhető
- Statikus metódusok
  - Csak az adott osztály statikus tagjait használhatja
  - Nem hivatkozhat a `this`-re
  - Nem lehet virtuális vagy konstans (`const`)

Statikus tagok

Macskák megszámolása

- Egészítsük ki a korábbi `Macska`-s példát úgy, hogy számoljuk, hogy hány macska lett létrehozva
  - Fel kell venni egy statikus adattagot, amely tárolja a példányosítások számát
    - Nem objektumonkét kell egy ilyen számláló, hanem egy közös kell
  - Kezdetben 0 az értéke
  - Minden példányosításnál növelni kell eggyel
- Lekérés
  - A statikus tagokra is érvényes a láthatóság
  - A darabszám lekéréshez szükség van egy statikus metódusra
    - A statikus adattaghoz hozzáférhet

Statikus tagok - demo (static 2)

Macskák megszámolása (folyt.)

```
class Macska {
  string nev = "Cirmi";
  static unsigned darab;
public:
  Macska() {
    ++darab;
  }

  Macska(const string& n) : nev(n) {
    ++darab;
  }

  void jatszik() const {
    // a macska jatszik ...
  }

  static unsigned getDarab() {
    return darab;
  }
};

unsigned Macska::darab = 0;
```

```
int main() {
  Macska k1("Morzsi");
  Macska k2("Foltos");
  k1.jatszik();
  k2.jatszik();
  k2.jatszik();
  k2.jatszik();
  cout << "Macskak szama: "
       << Macska::getDarab() << endl;
}
```

- Kimenet: 2

Statikus tagok inicializálása

- Osztályon kívül adhatunk értéket

```
class Macska {
  static int darab;
};

int Macska::db = 0;
```

- C++17 óta van másik megoldás

```
class Macska {
  static inline int darab = 0;
};
```

- Konstans esetében

```
class Macska {
  static const int maxDarab = 100;
};
```

Globális objektumok - demo (global)

```
class Nyilvantarto {
  vector<Macska> macskak;
public:
  Nyilvantarto() {
    cout << "A nyilvantartas elindult" << endl;
  }

  ~Nyilvantarto() {
    cout << "A nyilvantartas lezarult" << endl;
    for (const auto& m : macskak)
      cout << "- " << m << endl;
  }

  Nyilvantarto& operator+=(const Macska& macska) {
    macskak.push_back(macska);
    return *this;
  }
};

Nyilvantarto nyilvantarto;

int main() {                                    // kimenet
  cout << "A main elindult" << endl;            //   A nyilvantartas elindult
                                                //   A main elindult
  Macska cirmi;                                 //   A main veget ert
  Macska foltos("Foltos");                      //   A nyilvantartas lezarult
  nyilvantarto += cirmi;                        //   - Cirmi
  nyilvantarto += foltos;                       //   - Foltos

  cout << "A main veget ert" << endl;
}
```

Globális objektumok (folyt.)

- A globális objektumok valamely névtérben találhatóak
  - Nem osztály adattagok
  - Nem metódus blokkban vannak
- A program indulásakor létrejönnek
  - Hamarabb, mint ahogy a `main` végrehajtása elkezdődik
- Inicializálás
  - A nem `const` változók (obj.) automatikusan inicializálva lesznek

```
int x; // x erteke 0
```

  - A `const` globális változókat nekünk kell inicializálni
- Inicializálás sorrendje
  - Fordítási egységen belül a definiálás sorrendje
  - Különböző fordítási egységek esetében nem definiált
- A program kilépésekor megszűnnek
  - Desturktorok meghívódnak

Tartalom

Gyakorlat menetrend

- 2. ZH folyamatban
  - Van-e, aki igazoltan hiányzott, de pénteken pótolna?
- Pótlás, javítás: nov. 21. (péntek) 16 óra
  - Jelentkezési határidő: nov. 20. (csütörtök) 10 óra
    - Aki nem jelentkezik, nem javíthat
- Igazolt hiányzóknak javítás: nov. 25. (kedd) 18 óra
  - Csak az jöhet, aki igazoltan hiányzott, de szeretne javítani
- Csak 1 javítás lehet
  - Aki már javított korábban, az nem javíthat többször
    - Aki jelentkezett, nem jött el, de igazolása sincs, az elhasználta a javítását
- Igazolatlan hiányzás esetén **nem értékelhető** a félév
  - Ha valaki javítani szeretne a vizsgaidőszakban, írja meg mind a 3 ZH-t

Egyéb gyakorlattal kapcsolatos észrevételek

- Házi (gyakorló) feladatok
  - Érdemes megcsinálni „önállóan”
- „Anomália”
  - Házi elsőre max. pont
  - ZH 0 pont
- Nagyon kell még plusz pont a gyakorlaton a jobb jegyhez. Lehet még szerezni?
  - Nov. 11-én a válasz: igen, gyakorlaton lehet plusz pontokat szerezni
    - Concierge regisztráció
  - **3. ZH megírása után: NEM**

Tartalom

Implicit konverzió

- Implicit konverzió akkor hajtódik végre, ha egy értéket egy vele kompatibilis értékké konvertálunk
  - Példa, amikor `int`-et adunk értékül egy `double` típusnak

```
int i = 5;
double d = i;
```

- Ilyenkor „automatikusan” végrehajtódik a konverzió

Standard konverzió - alap típusok

- Az alap típusok közötti konverzió
  - Numerikus típusok között
    - `short` → `int`, `int` → `double`, `float` → `long`
  - Logikai és numerikus típus között
    - numerikus ↔ `bool`
- Promotion
  - Ha kisebb egész típusról egy nagyobb egész típusra konvertálunk, akkor garantáltan ugyanazt az értéket kapjuk az új típus esetén is
    - Például: `short` → `int` → `long`
  - Ez igaz valós számokra is
    - Például: `float` → `double`
- Numerikus típusok közötti egyéb konverzió értékvesztéssel járhat
  - Például ha az új típus értéktartományán kívül esik a reprezentálandó érték

Standard konverzió - alap típusok

„Veszteségek, mellékhatások”

- Előjeles és előjeltelen egészek közötti konverzió
  - Kettes komplemens segítségével ábrázoljuk a negatív számokat
  - Ha a -1-et konvertáljuk előjeltelenné, akkor a legnagyobb számot kapjuk
  - Ha az előjeltelen „nagy” számokat konvertáljuk előjelessé, akkor meg negatív számokat kapunk
- Logikai típus konvertálása
  - A hamis érték (`false`) egész típusként a 0-val egyenlő, míg pointerként null pointerrel egyenlő
    - Oda-vissza konvertálható
  - Az igaz érték (`true`) ekvivalens minden nem 0 értékkel, és 1-re konvertálódik
- Ha valós típusról egész típusra konvertálunk, akkor a decimális részt levágjuk (és nem kerekítjük)
  - Ha az így kapott érték kívül esik az egész típus értéktartományán, akkor a konverzió nem definiált viselkedést eredményez

Standard konverzió - nem-alap típusok

- A pointerek implicit konverziója
  - A null pointer (`nullptr`) bármilyen pointer típusra konvertálható
  - Bármilyen típusú pointer `void` pointerré konvertálható
  - Egy leszármazott osztályra mutató pointer konvertálható ősosztályra mutató pointerré
    - Ha a konverzió *hozzáférhető* és *egyértelmű*
    - Nem változtatja meg a `const` és `volatile` módosítókat
- `int *p = NULL;`
  - C++11 előtt 0 volt
  - Azóta `nullptr`

Implicit konverzió osztályok esetében

- Osztályok esetében az implicit konverziók
  - Egy paraméteres konstruktor
    - Lehetővé teszi az implicit konverziót az adott típusról az osztály típusára objektum lértehozásakor
  - Értékadó operátor
    - Lehetővé teszi az implicit konverziót az adott típusról az osztály típusára értékadáskor
  - Konverziós operátor
    - Lehetővé teszi az implicit konverziót az adott típusra

```
class MyNum {
  int i = 0;
public:
  MyNum(int n) : i(n) { }       // konstruktor
  MyNum& operator=(int n) {     // ertekado operator
    i = n;
    return *this;
  }
  operator int() { return i; } // konverzios operator
};
```

Implicit konverzió „mellékhatásai”

Példa

```
class Compl {
  double real, imag;
public:
  Compl(double r, double i) : real(r), imag(i) { }

  Compl(double r) : Compl(r, 0) { }

  operator double() const { return real; }
};

void komplex(Compl c) { /* ... */ }
void valos(double d) { /* ... */ }

int main() {
  Compl c(3, 6);
  komplex(3); // Compl tipust var, de int-tel hivjuk
  valos(c);   // double tipust var, de Compl-lal hivjuk
}
```

explicit konverzió

- A konstruktor, egyenlőség vagy konverziós operátor által történő automatikus konverzió nem mellékhatás
  - Ez az elvárt működése
  - „Mellékhatás”: nem számítunk rá, hibákat okozhat, . . .
- Megakadályozhatjuk, hogy automatikus legyen
  - Az `explicit` kulcsszó segítségével
    - A konstruktor és a konverziós operátor lehet `explicit`

```
class Compl {
  // ...
  explicit Compl(double r);
  explicit operator double();
};
void komplex(Compl c);
void valos(double d);
int main() {
  Compl c(3, 6);
  komplex(3); // HIBA!
  valos(c);   // HIBA!
}
```

```
class Compl {
  // ...
  explicit Compl(double r);
  explicit operator double();
};
void komplex(Compl c);
void valos(double d);
int main() {
  Compl c(3, 6);
  komplex((Compl)3); // helyes
  valos((double)c);  // helyes
}
```

Tartalom

C-szerű típus konverzió

- Lehetőség van explicit típus konverziót is alkalmazni
  - Megadhatjuk, hogy milyen típusra akarjuk konvertálni
- C-ben is van típus kényszerítés

```
double d = 3.14;
int i = (int) d;
```

  - Ezt lehet C++-ban is használni
    - C++-ban ezt C-szerű típus konverziónak hívják
- C++-ban lehet függvény-szerűen is írni

```
int i = int (d);
```

  - Ez a két típus konverzió megegyezik

C-szerű típus konverzió

- A C-szerű típus konverzióval bármilyen pointer típust bármilyen pointer típussá lehet konvertálni
  - Ez veszélyes és hibához vezethet

```
class Kutya { /* ... */ };
class Macska { /* ... */ };

int main() {
  Kutya *kutya = new Kutya;
  Macska *macska = (Macska*)kutya;
}
```

  - Az ilyen konverziók hibához vezetnek
    - Futás közbeni hiba
    - Rossz működés, nem várt eredmény, stb.

Tartalom

C++ típus konverziók

- Az osztályok közötti típus konverziókra 4 konverziós operátor létezik
  - `dynamic_cast <uj_tipus> (kifejezes)`
  - `static_cast <uj_tipus> (kifejezes)`
  - `reinterpret_cast <uj_tipus> (kifejezes)`
  - `const_cast <uj_tipus> (kifejezes)`
- C-típusú konverzióval kifejezve ezeket a konverziókat
  - `uj_tipus (kifejezes)`
  - `(uj_tipus) kifejezes`
- A C++ típus konverziók előnyei
  - Többek, mint a C-típusú konverziók
    - Fordítás és futás közbeni ellenőrzés
  - Intuitív, tudjuk miért alkalmazzuk a konverziót
    - Különböző okai lehetnek, és az operátor sugallja ezt

dynamic_cast

- A dinamikus konverziót csak osztályokra mutató pointerrel és referenciával lehet használni
- Egymással öröklődési kapcsolatban álló osztályok közötti konverzióra használható
  - Leszármazott osztály ősosztályra való konvertálása
  - Ősosztály leszármazott osztályra történő konvertálása
    - Ha van polimorfizmus
    - És az objektum megfelelő típusú
- Futás közbeni ellenőrzést végez
  - Ha nem hajtható végre a konverzió, akkor null pointerrel tér vissza
- Használatához Run-Time Type Information (RTTI) szükséges
  - Futás közben nyilvántartja a dinamikus típusokat
  - Letiltható (pl. GCC esetében a -fno-rtti paraméterrel), de akkor nem használható a `dynamic_cast`
    - Fordítási hibát kapunk

dynamic_cast

Példa

```
class Emlos {
public:
  virtual void jatszik() const noexcept { /* ... */ };
};

class Macska: public Emlos {
public:
  virtual void jatszik() const noexcept override { /* ... */ }
};

int main () {
  // statikus tipus: Emlos, din. tipus: Macska
  Emlos *macska = new Macska;

  // statikus tipus: Emlos, din. tipus: Emlos
  Emlos* emlos = new Emlos;

  Macska* m1 = dynamic_cast<Macska*>(macska); // sikeres
  if (!m1)
    cout << "Sikertelen az elso type-cast" << endl;

  Macska* m2 = dynamic_cast<Macska*>(emlos); // sikertelen
  if (!m2)
    cout << "Sikertelen a masodik type-cast" << endl;
}
```

static_cast

- Egymással öröklődési kapcsolatban álló osztályra mutató pointerekre alkalmazható
  - Ősosztályról leszármazott osztályra és fordítva is lehet alkalmazni
- Fordítási időben történik az ellenőrzés
  - Gyorsabb, mert nincs futásidejű ellenőrzés
- Az ellenőrzés hiánya miatt lehet rosszul is használni
  - **A fejlesztő felelőssége, hogy helyesen használja**

static_cast

Példa

```
class Emlos {
public:
  virtual void jatszik() const noexcept { /* ... */ };
};

class Macska: public Emlos {
  string nev;
public:
  Macska(const string& nev) : nev(nev) { }
  virtual void jatszik() const noexcept override final { /* ... */ }
  void eszik() { cout << nev << " eszik" << endl; }
};

class Allatorvos { /* ... */ };

int main () {
  Emlos* emlos = new Emlos;            // statikus tipus: Emlos, din. tipus: Emlos
  Emlos* macska = new Macska("Cirmi"); // statikus tipus: Emlos, din. tipus: Macska

  // lefordul, helyes
  Macska* m = static_cast<Macska*>(macska);
  m->eszik();

  // lefordul, de NEM helyes
  Macska* e = static_cast<Macska*>(emlos);
  e->eszik();

  // forditasi hiba
  Allatorvos* orvos = static_cast<Allatorvos*>(emlos);
}
```

static_cast további használatai

- Egész vagy valós számot, illetve enumot lehet enum-ra konvertálni

```
enum E { valami };
E e = static_cast<E>(6);
```

- `void*` típust lehet tetszőleges pointerré konvertálni

```
int *p = static_cast<int*>(malloc(sizeof(int)));
```

reinterpret_cast

Pointerek között

- Bármilyen pointert bármilyen pointerré lehet konvertálni
  - Nincs semmilyen ellenőrzés
  - Az alábbi példa lefordul
    - De hibához vezet

```
class Macska { /* ... */ };
class Allatorvos { /* ... */ };

int main () {
  Macska *macska = new Macska;
  Allatorvos *orvos = reinterpret_cast<Allatorvos*>(macska);
}
```

reinterpret_cast

Pointer és egész szám között

- Bármilyen egész szám pointerré konvertálható

```
class Pelda { /* ... */ };
Pelda *p = reinterpret_cast<Pelda*>(800);
```

- Bármilyen pointer egész számmá konvertálható

```
class Pelda { /* ... */ };
Pelda *p = new Pelda;
int i = reinterpret_cast<int>(p);
```

const_cast

- Az adott objektum `const` tulajdonságát távolítja el
  - Nem okoz hibát, ha nem `const`-ra alkalmazzuk
- A `const` módosítót szándékosan használják
  - Arra utalnak, hogy azt nem szabad megváltoztatni
  - Ha eltávolítjuk, legyünk biztosak benne, hogy nem okoz gondot

```
void to_lower(char* str) {
  for ( ; *str; ++str)
    if ('A' <= *str && *str <= 'Z')
      *str += 'a' - 'A';
}

int main() {
  const char *str = "Pelda";
  to_lower(const_cast<char*>(str)); // hibahoz vezet!
  cout << str << endl;
}
```

const_cast

Példa

```
class Szemely {
  string name;
public:
  Szemely(string n) : name(n) { }
  const string& getName() const {
    return name;
  }
};

int main () {
  Szemely szemely("Kiss Endre");
  cout << szemely.getName() << endl;
  szemely.getName() = "Nagy Geza"; // forditasi HIBA!
  const_cast<string&>(szemely.getName()) = "Nagy Geza";
  cout << szemely.getName() << endl;
}
```

Tartalom

Iteratorok

- Iterátorok „általános bejárást” biztosítanak a tárolókon
  - Megvalósíthatunk saját algoritmusokat
    - ++, !=, ==, =, *, -> operátorok értelmezve vannak rá

Iteratorok

- Iterátorok „általános bejárást” biztosítanak a tárolókon
  - Megvalósíthatunk saját algoritmusokat
    - ++, !=, ==, =, *, -> operátorok értelmezve vannak rá
- Probléma
  - Milyen típusokat tárolunk
    - `vector<int>` és `vector<long>` különböző típusok
  - Különböző tárolókhoz különböző iterátor tartozik
    - `vector<int>` és `set<int>` iterátora különböző
- Megoldás: típus-független algoritmusok
  - Elegendő, hogy a típusra értelmezve vannak a szükséges operátorok
  - Azt is megadhatjuk, hogy milyen műveletet szeretnénk elvégezni

Algoritmusok

https://en.cppreference.com/w/cpp/algorithm

- `for_each`

```
template<class InputIt, class UnaryFunct>
UnaryFunction for_each(InputIt first, InputIt last, UnaryFunct f)
```

- Működése
  - A first-től a last-ig végig iterál az elemeken
  - Minden elemre „végrehajtja az f műveletet”
- A paraméter típusa lehet
  - Függvény
  - Függvény objektum
  - Lambda

```
void kiir(int i) { cout << i << ", "; }

bool paros(int i) { return i % 2 == 0; }

int main() {
  vector<int> v {1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  for_each(v.begin(), v.end(), kiir);
  cout << endl;

  cout << "paros db: " << count_if(v.begin(), v.end(), paros) << endl;
}
```

Nem módosító algoritmusok

Példák

- `for_each`
  - Minden egyes elemre végrehajtja a műveletet
- `count`, `count_if`
  - Megszámolja, hogy hány olyan elem van, ami eleget tesz a feltételnek
    - `count` esetében egyenlő az adott elemmel
    - `count_if` esetében igazzal tér vissza a predikátum
- `find`, `find_if`
  - Megkeresi az adott elemet
    - `find` esetében egyenlő az adott elemmel
    - `find_if` esetében igazzal tér vissza a predikátum
- `search`
  - Adott elemekből álló intervallum keresése

Módosító algoritmusok

Példák

- `copy`, `copy_if`
  - Elemek másolása
    - `copy_if` esetében csak akkor lesz másolva, ha a predikátum igaz
- `fill`, `fill_n`
  - Adott intervallumot inicializál az elemmel
    - `fill` esetében a két iterator közötti elemeket
    - `fill_n` esetében darabszámot adhatunk meg

Algoritmusok használata

Nehézségek, hátrányok

- Az algoritmusokhoz olyan függvényeket használunk, amik
  - „egyediek”, máshol nem tudjuk felhasználni
    - felesleges ezért egy függvényt írni
  - „általánosak”, például egy értéknél nagyobbakra legyen igaz
    - van már rá megoldás
  - „egyszerűek”, pár utasításból állnak
    - nem is szerveznénk ki függvénybe
  - használják a lokális változókat
    - át kell adni azokat
  - távol vannak a felhasználástól
    - nehezebb megérteni a kódot

Lambda

```
for_each(v.begin(), v.end(), [](int i) { cout << i << ", "; } );
```

- [] captures
  - Megadhatjuk, hogy a „környezetéből” mit és hogyan érjen el
- (...) paraméter lista
  - Ezekkel a paraméterekkel lesz meghívva
- {} lambda törzse
  - Ez fog végrehajtódni

```
int main() {
  vector<int> v{1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
  vector<int> w(v.size());
  set<int> s;

  copy(v1.begin(), v.end(), w.begin());
  copy_if(v.begin(), v.begin()+5, back_inserter(w), [](int i){ return i % 2 != 0; });
  copy_if(v.crbegin(), v1.crend(), inserter(s, s.end()), [](int i){ return i < 4; });
}
```

Lambda captures

- `[]`
  - Semmit sem használ a környezetéből
- `[limit]`
  - Érték szerint veszi át a változót

```
[limit](int i) { if (limit < i ) cout << i << ", "; }
```

- `[&limit]`
  - Referencia szerint veszi át a változót
    - Ha megváltoztatom, akkor az „eredeti érték” is megváltozik

```
[&limit](int i) { if (limit++ < i ) cout << i << ", "; }
```

- Több paraméter: `[&lower, upper]`
  - Külön megadhatjuk, hogy melyik változót hogyan használunk

```
[&lower, upper](int i) {
  if (lower++ < i && i < upper) cout << i << ", ";
}
```

Lambda captures

- [=]
  - Összes változót érték szerint veszi át

```
[=](int i) { if (limit < i ) cout << i << ", "; }
```

- [&]
  - Összes változót referencia szerint veszi át

```
unsigned sum = 0, db = 0, limit = 5;
[&](unsigned i) {
  ++db;
  if (lower < i)
    sum += i;
}
```

- [=, &db]
  - Mindent érték szerint, kivéve ...

```
[=, &db](int i) { db++; if (limit < i ) cout << i << ", "; }
```

- [&, limit]
  - Mindent referencia szerint, kivéve ...

```
[&, limit](int i) { if (limit < i ) ++db; }
```

Lambda példák

- Érték szerint átvétel

```
int limit = 4;
cout << limit << " nagyobb ertekek szama: ";
cout << count_if(v.begin(), v.end(), [limit](int i) { return limit < i; } );
```

- Referencia szerinti átvétel

```
int sum = 0;
for_each(v.begin(), v.end(), [&sum](int i) { sum += i;} );
cout << "sum = " << sum << endl;
```

- Érték szerint több, vagy akár az összes átvétele

```
int min = 3, max = 5;
count_if(v.begin(), v.end(), [min, max](int i) { return min < i && i < max; } );
count_if(v.begin(), v.end(), [=](int i) { return min < i && i < max; } );
```

- Érték és referencia szerint átvétel

```
int sum = 0;
int min = 3, max = 7;
for_each(v.begin(), v.end(), [=, &sum](int i){ if (min < i && i < max) sum+=i; });
cout << sum << endl;
```

Lambda példák (folyt.)

- A paraméter is lehet referencia

```
int sum = 0, min = 5;
for_each(v2.begin(), v2.end(), [min, &sum](int &i) {
  sum += i;
  if (i < min)
    i = min;
} );
```

- `this` átvétele

```
class MyLimit {
  int min = 0, max = 10;
  const vector<int> v;
public:
  void kiir(int pmin, int pmax) {
    for_each(v.begin(), v.end(), [=, this](int i) {
      this->min = pmin;
      this->max = pmax;
      if (this->min < i && i < this->max)
        cout << i << ", ";
    });
    cout << endl;
  }
  void kiir() {
    for_each(v.cbegin(), v.cend(), [*this](int i) {
      if (this->min < i && i < this->max)
        cout << i << ", ";
    });
  }
};
```

Tartalom

Dinamikus memória kezelése C++-ban

Problémák

- A kupacon (heap) „könnyű” memóriát foglalni
  - `new`, `new[]`, `malloc`, ...
- Mikor szabadítsuk fel?
  - Ha nem szabadítjuk fel, akkor szivárog a memória
  - Ha „túl korán” szabadítjuk fel, de még használnánk, akkor UB
- Le van még foglalva?
  - Egyik pointeren keresztül felszabadítjuk, a másik pointer nem tud erről
    - Ha használjuk az objektumot, akkor UB
- Java sokkal jobb volt ☺

Okos pointerek

- C++-ban is megoldható, hogy az objektumok megszűnjenek, ha már „nem használjuk” azokat
  - Csak akkor szűnjenek meg
  - Biztosan megszűnjenek
- Wrapper osztály
  - Tárolja a pointert a lefoglalt memóriára
  - Amíg a wrapper osztály él, addig elérjük az objektumot
  - Amikor az utolsó wrapper osztály is megszűnik, akkor törli az objektumot
- Honnan tudjuk, hogy ki az utolsó?
  - Számoljuk hány van belőle
  - Csak egy lehet belőle

shared_ptr

- Okos pointer, ami egy mutatót segítségével „eltárolja” az objektumot
  - Egy pointert tárol az adott objektumra
  - Nem kizárólagos tulajdonosa az objektumnak
    - Több `shared_ptr` is „tárolhatja” ugyanazt az objektumot
- Az utolsó megszünteti a tárolt objektumot
  - Az utolsó `shared_ptr` megszűnik
  - Az utolsó `shared_ptr` másik értéket kap
- Úgy „viselkedik”, mint egy pointer
  - Megadhatunk egy pointer létrehozáskor
    - A konstruktor mellett a `make_shared` is használható
  - Lehet `nullptr` az értéke
  - Meg van valósítva a `*` és a `->` operátora
- További funkcionalitások
  - `get()` - visszaadja a pointert
  - `use_count()` - megmondja, hány különböző `shared_ptr` menedzseli az adott objektumot

shared_ptr példa

```
class Allat {
  string faj;
public:
  Allat(const string& faj) : faj(faj) { }
  virtual ~Allat() { cout << "Allat megszunik" << endl; }
  virtual void print() const { cout << "Allat: " << faj << endl; }
};

class Macska : public Allat {
  string nev;
public:
  Macska(const string& nev) : Allat("macska"), nev(nev) { }
  ~Macska() override { cout << "Macska megszunik, "; }
  void print() const override { cout << "Macska: " << nev << endl; }
};

int main() {
  shared_ptr<Allat> allatPtr;
  {
    shared_ptr<Macska> macskaPtr(new Macska("Cirmi"));
    allatPtr = macskaPtr;
    cout << "use_count: " << allatPtr.use_count() << endl;
    cout << "blokk vege" << endl;
  }
  cout << "use_count: " << allatPtr.use_count() << endl;

  allatPtr->print();

  cout << "main vege" << endl;
}
```

shared_ptr hibák

- Nem a kupacon létrehozott objektummal inicializáljuk

```
class Menhely {
  vector<shared_ptr<Allat>> allatok;
public:
  void add (Allat* ptr) {
    allatok.push_back(shared_ptr<Allat>(ptr));
  }
};

int main() {
  Macska macska("Cirmi");
  Menhely().add(&macska); // HIBA!!!
}
```

- Egy objektumot két „egymásról nem tudó” `shared_ptr` tárol

```
shared_ptr<Macska> macska1(new Macska("Cirmi"));
shared_ptr<Macska> macska2(macska1.get()); // HIBA!!!
```

- `shared_ptr<Macska>` nem öröklődik `shared_ptr<Allat>`-ból

```
class Allat {
  virtual shared_ptr<Allat> clone() const { return make_shared<Allat>(*this); }
};

class Macska : public Allat {
  shared_ptr<Macska> clone() const override { return make_shared<Macska>(*this); }
};
```

shared_ptr helyesen

- Nem a kupacon létrehozott objektummal inicializáljuk

```
class Menhely {
  vector<shared_ptr<Allat>> allatok;
public:
  void add (Allat* ptr) {
    allatok.push_back(shared_ptr<Allat>(ptr));
  }
};

int main() {
  Macska* macska = new Macska("Cirmi");
  Menhely().add(macska);
}
```

- Egy objektumot két „egymásról nem tudó” `shared_ptr` tárol

```
shared_ptr<Macska> macska1(new Macska("Cirmi"));
shared_ptr<Macska> macska2(macska1); // HIBA!!!
```

- `shared_ptr<Macska>` nem öröklődik `shared_ptr<Allat>`-ból

```
class Allat {
  virtual shared_ptr<Allat> clone() const { return make_shared<Allat>(*this); }
};

class Macska : public Allat {
  shared_ptr<Allat> clone() const override { return make_shared<Macska>(*this); }
};
```

unique_ptr

- Okos pointer, ami egy mutatót segítségével „eltárolja” az objektumot
  - **Kizárólagos** tulajdonosa az objektumnak
    - Csak egy `unique_ptr` „tárolhatja” az adott objektumot
- Amikor megszűnik, akkor megszünteti a tárolt objektumot
  - A `unique_ptr` megszűnik vagy másik értéket kap
- Úgy „viselkedik”, mint egy pointer
  - Megadhatunk egy pointer létrehozáskor (`make_unique`)
  - Lehet `nullptr` az értéke
  - Meg van valósítva a $*$ és a $->$ operátora
- További funkcionalitások
  - `get()` - visszaadja a pointert
- **Nem másolható**
  - `operator=(const unique_ptr&)` törölve van
  - `unique_ptr(const unique_ptr&)` törölve van
- „Átadható” a tárolt objektum
  - `ptr1 = std::move(ptr2);`
    - ptr1 átveszi a ptr2 által tárolt objektumot
    - ptr2 `nullptr` lesz

unique_ptr példa

```
class Allat {
  string faj;
public:
  Allat(const string& faj) : faj(faj) { }
  virtual ~Allat() { cout << "Allat megszunik" << endl; }
  virtual void print() const { cout << "Allat: " << faj << endl; }
};

class Macska : public Allat {
  string nev;
public:
  Macska(const string& nev) : Allat("macska"), nev(nev) { }
  ~Macska() override { cout << "Macska megszunik, "; }
  void print() const override { cout << "Macska: " << nev << endl; }
};

int main() {
  unique_ptr<Allat> allatPtr;
  {
    unique_ptr<Macska> macskaPtr(new Macska("Cirmi"));
    macskaPtr->print();
    allatPtr = std::move(macskaPtr);
    if (macskaPtr.get())
      macskaPtr->print(); // nem fog kiirodni
    cout << "blokk vege" << endl;
  }

  allatPtr->print();

  cout << "main vege" << endl;
}
```

unique_ptr hibák

- Nem másolható

```
unique_ptr<Macska> macskaPtr(new Macska("Cirmi"));
unique_ptr<Allat> allatPtr = macskaPtr; // HIBA!!!
```

- Paraméterátadás, `unique_ptr`-t tartalmazó osztály másolása, ...

```
class Menhely {
  vector<unique_ptr<Allat>> allatok;
public:
  void add (unique_ptr<Allat> allat) {
    allatok.push_back(allat); // HIBA!!!
  }
};

int main() {
  Menhely menhely;
  unique_ptr<Allat> macska(new Macska("Cirmi"));
  menhely.add(macska); // HIBA!!!

  Menhely menhely2;
  menhely2 = menhely; // HIBA!!!
}
```

Helyes használat

- `move` használata

```
class Menhely {
  vector<unique_ptr<Allat>> allatok;
public:
  void add (unique_ptr<Allat> allat) {
    allatok.push_back(move(allat));
  }
};

int main() {
  Menhely menhely;
  unique_ptr<Allat> macska(new Macska("Cirmi"));
  menhely.add(move(macska));
}
```

- Temporáris objektum

```
menhely.add(unique_ptr<Macska>(new Macska("Cirmi")));
menhely.add(make_unique<Macska>("Cirmi"));
```

- Referencia paraméter (nem másolódik)

```
void addRef (unique_ptr<Allat>& allat) {
  allatok.push_back(move(allat));
}

unique_ptr<Allat> macska(new Macska("Cirmi"));
menhely.addRef(macska);
```

Tartalom

Kivételkezelés célja

- A program futása során történhetnek kivételes események, futási hibák, amik a normál működést akadályozzák
  - Megszakad a hálózati kapcsolat
  - Elfogy a memória vagy a merevlemezen a hely
  - Nincs írási jogunk egy mappára
- Ezek kivételes esetek
  - Ezekre nem akarunk vagy nem tudunk felkészülni
- C++-ban megjelent a magasszintű kivételkezelési mechanizmus
  - Javahoz hasonló, de vannak eltérések
- Előnyök
  - Különválik a tényleges kód és a hibakezelés
  - Szétoszthtató a felelősség
    - Minden metódus megválogathatja, hogy milyen kivételeket képes az adott ponton lekezelni
    - Ott kezeljük a hibát, ahol tudjuk
  - Kivétel osztályokból is készíthető hierarchia, öröklődés, stb.

Kivételkezelés C++-ban

- A kivételek kezelését néhány, már ismert kulcsszó segítségével tehetjük meg
  - `throw` segítségével dobhatunk kivételt
  - `try` a védett régió
    - A blokk, amiben keletkezhet kivétel
  - `catch` kivételt kezelő kódrészlet
    - Több is lehet belőle
  - `noexcept` segítségével a függvény fejlécében jelezhetjük, hogy a függvény dobhat-e kivételt
    - `throw()`: függvény fejlécében jelezhetjük, hogy a függvény milyen típusú kivételeket dobhat

Kivételkezelés példa

```
class Datum {
  const unsigned ev, honap, nap;
public:
  Datum(unsigned ev, unsigned honap, unsigned nap) :
    ev(ev), honap(honap), nap(nap)
  {
    unsigned maxNapok = 31;
    switch (honap) {
      case 2:
        maxNapok = ((ev%4==0 && ev%100!=0) || ev%400==0) ? 29 : 28;
        break;
      case 4: case 6: case 9: case 11:
        --maxNapok;
      case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        break;
      default:
        // kivetel dobasa
        throw out_of_range("rossz honap: " + to_string(honap));
    }
    if (maxNapok < nap || nap == 0)
      // kivetel dobasa
      throw out_of_range("rossz nap: " + to_string(nap));
  }

  friend ostream& operator<<(ostream& os, const Datum& d) {
    return os << d.ev << "-" << d.honap << "-" << d.nap << endl;
  }
};
```

```
try {
  cout << Datum(2024, 10, 22);
  cout << Datum(2024, 22, 30);
} catch (const out_of_range& e) {
  cout << e.what() << endl;
} catch (const exception& e) {
  cout << e.what() << endl;
}
```

Kivétel dobása

- Van egy kódunk, amiben egy kivételes esemény következik be
  - Ekkor a `throw` segítségével dobhatunk egy kivételt
- Kivételek dobásakor „tetszőleges” típusú kivételt dobhatunk, **DE!**
  - `int`, `bool`, `char`, ...
  - Egyszerű típusok helyett célszerű „kivétel objektumokkal” dolgozni
    - Nem csak egy számot tudunk „üzenni”
    - Hierarchiába tudjuk szervezni
- `std::exception` közös ősosztály a kivételekhez
  - Használatukhoz szükséges az `<exception>` header
    - Ezt követően használhatjuk a beépített `std::exception` osztályt
  - `virtual const char* what() const noexcept;`
    - Magyarázó string (C-string)
  - Beépített kivételek
    - `https://en.cppreference.com/w/cpp/error/exception`
  - Ezeket használják a „beépített” függvények, operátorok, ...
    - `at: out_of_range`
    - `stoi: invalid_argument, out_of_range`

Kivétel elkapása

- Több kivételt is el lehet kapni
  - Az kapja el, amelyikre a sorban elsőre ráilleszkedik

```
int konvertal(const string& str) {
  try {
    return stoi(str); // ketfele kivetelt dobhat
  } catch (const invalid_argument& e_inv) {
    cerr << "invalid_argument exception" << endl;
    return -1;
  } catch (const out_of_range& e_out) {
    cerr << "out_of_range exception" << endl;
    return INT_MAX;
  }
}

int main() {
  cout << konvertal("42") << endl;
  cout << konvertal("aaa") << endl;
  cout << konvertal("999999999999999") << endl;
}
```

Akkor kapjuk el, ha kezelni is tudjuk

- Ott kapjuk el a kivételt, ahol tudjuk, hogy hogyan kell kezelni
  - Függvény esetében nem kell feltüntetni, hogy milyen kivételt dobhat
    - A kezeletlen kivétel elhagyhatja a függvényt

```
int konvertal(const string& str) { // barmilyen kivetelt dobhat
  try {
    return stoi(str); // ketfele kivetelt dobhat
  } catch (const out_of_range& e_out) {
    cout << "out_of_range exception" << endl;
    return INT_MAX;
  }
}

int main() {
  cout << konvertal("42") << endl;
  try {
    cout << konvertal("aaa") << endl;
  } catch (const invalid_argument& e_inv) { // masik kivetel elkapasa
    // hibakezeles
    cerr << "invalid_argument exception" << endl;
  }
  cout << konvertal("999999999999999") << endl;
}
```

„Saját kivétel osztály”

```
class Kivetel { // sajat kivetel osztaly, nem kell az std::exception-bol szarmaznia
  string msg;
public:
  Kivetel(const string& s) : msg(s) { }

  const string& uzenet() const { return msg; }
};

class Datum {
  const unsigned ev, honap, nap;
public:
  Datum(unsigned ev, unsigned honap, unsigned nap) :
    ev(ev), honap(honap), nap(nap)
  {
    unsigned maxNapok = 31;
    switch (honap) {
      case 2:
        maxNapok = ((ev%4==0 && ev%100!=0) || ev%400==0) ? 29 : 28;
        break;
      case 4: case 6: case 9: case 11:
        --maxNapok;
      case 1: case 3: case 5: case 7: case 8: case 10: case 12:
        break;
      default:
        throw Kivetel("rossz honap: " + to_string(honap)); // kivetel dobasa
    }
    if (maxNapok < nap || !nap)
      throw Kivetel("rossz nap: " + to_string(nap)); // kivetel dobasa
  }
};
```

Kivételek öröklődési hierarchiába szervezése

- Lehetőségünk van a kivételeket is származtatni egymásból

```
class Kivetel {
  string msg;
public:
  Kivetel(const string& s) : msg(s) { }

  virtual const string& uzenet() const {
    return msg;
  }
};

class RosszHonap : public Kivetel {
public:
  RosszHonap(int h) :
    Kivetel("rossz honap: "+to_string(h))
    { }
};

class RosszNap : public Kivetel {
public:
  RosszNap(int n) :
    Kivetel("rossz nap: "+to_string(n))
    { }
};
```

```
class Datum {
  const unsigned ev, honap, nap;
public:
  Datum(/* ev, honap, nap */) {
    switch (honap) {
      /* ... */
      default:
        throw RosszHonap(honap);
    }
    if (maxNapok < nap || !nap)
      throw RosszNap(nap);
  }
};

int main() {
  try {
    Datum d1(2024, 10, 22);
    Datum d2(2024, 22, 30);
  } catch (const RosszHonap& e) {
    // honap hiba kezelese
    cout << e.uzenet() << endl;
  } catch (const RosszNap& e) {
    // nap hiba kezelese
    cout << e.uzenet() << endl;
  } catch (const Kivetel& e) {
    // altalanos hiba keyelese
    cout << e.uzenet() << endl;
  }
}
```

Mire figyeljünk a kivétel elkapásakor?

Sorrend

- Azonos típusú kivételeket őstípusként is el lehet kapni

```
try {
  Datum d(2024, 22, 30);
} catch (const RosszHonap& e) {
  cout << e.uzenet() << endl;
} catch (const RosszNap& e) {
  cout << e.uzenet() << endl;
} catch (const Kivetel& e) {
  cout << e.uzenet() << endl;
}
```

```
try {
  Datum d(2024, 22, 30);
} catch (const Kivetel& e) {
  cout << e.uzenet() << endl;
}
```

- Előbb a speciális kivételeket kell elkapni, aztán az általánosat
  - Az általános ráillik a speciálisra is

- Jó sorrend

```
try {
  Datum d(2024, 22, 30);
} catch (const RosszNap& e) {
  cout << e.uzenet() << endl;
} catch (const Kivetel& e) {
  // altalanos a spec. utan
  cout << e.uzenet() << endl;
}
```

- **Rossz sorrend**

```
try {
  Datum d(2024, 22, 30);
} catch (const Kivetel& e) {
  cout << e.uzenet() << endl;
} catch (const RosszNap& e) {
  // alt. kivetel elkapta, sosem fut le
  cout << e.uzenet() << endl;
}
```

Mire figyeljünk a kivétel elkapásakor?

Publikus öröklődés hiánya

- Ha nem publikus az öröklődés, akkor őstípus alapján nem lehet elkapni

```
class Kivetel { /* ... */ };

class RosszHonapKivetel : protected Kivetel { /* ... */ };

class RosszNapKivetel : protected Kivetel { /* ... */ };

class Datum {
  Datum(/* ev, honap, nap */) {
    /* ... */
      throw RosszHonap(honap);
    /* ... */
      throw RosszNap(nap);
  }
};

int main() {
  try {
    cout << Datum(2024, 10, 22) << endl;
    cout << Datum(2024, 22, 30) << endl;
  } catch (const Kivetel& e) { // nem kapja el, pedig ostipus
    cout << "Kiv.: " << e.uzenet() << endl;
  } catch (const RosszHonap& e) {
    cout << "Rossz honap kiv.: " << e.uzenet() << endl;
  } catch (const RosszNap& e) {
    cout << "Rossz nap kiv.: " << e.uzenet() << endl;
  }
}
```

Mire figyeljünk a kivétel elkapásakor?

Throw by value, catch by reference

- **Érték szerint dobjuk, referencia szerint kapjuk el**

```
class Kivetel {
public:
  virtual string uzenet() const { return "Kivetel"; }
};

class RosszNapKivetel : public Kivetel {
public:
  string uzenet() const override { return "RosszNapKivetel"; }
};

class Datum {
  Datum(/* ev, honap, nap */) {
    /* ... */
      throw RosszNap(nap);
  }
};
```

- **Referencia szerint elkapva**

```
try {
  Datum d(2024, 22, 30);
} catch (const Kivetel& k) {
  cout << "Hiba: " << k.uzenet() << endl;
  // Hiba: RosszNapKivetel
}
```

- **Érték szerint elkapva**

```
try {
    Datum d(2024, 22, 30);
} catch (const Kivetel k) {
  cout << "Hiba: " << k.uzenet() << endl;
  // Hiba: Kivetel
}
```

Kivétel továbbdobása

- A `throw;` segítségével ismét eldobhatjuk a kivételt

```
class Datum {
  const unsigned ev, honap, nap;
public:
  Datum(unsigned ev, unsigned honap, unsigned nap) :
    ev(ev), honap(honap), nap(nap)
  {
    try {
      unsigned maxNapok = 31;
      switch (honap) {
        case 2:
          maxNapok = ((ev%4==0 && ev%100!=0) || ev%400==0) ? 29 : 28;
          break;
        case 4: case 6: case 9: case 11:
          --maxNapok;
        case 1: case 3: case 5: case 7: case 8: case 10: case 12:
          break;
        default:
          throw RosszHonap(honap); // kiv. dobasa
      }
      if (maxNapok < nap || !nap)
        throw RosszNap(nap); // kivetel dobasa
    } catch(const Kivetel& k) {
      cout << "Log: " << k.uzenet() << endl;
      throw; // elkapott kivetel ismetelt dobasa
    }
  }
};
```

```
int main() {
  try {
    cout << Datum(2024, 22, 22);
  } catch (const RosszHonap& e) {
    cout << e.uzenet() << endl;
  } catch (const RosszNap& e) {
    cout << e.uzenet() << endl;
  } catch (const Kivetel& e) {
    cout << e.uzenet() << endl;
  }
}
```

Resource acquisition is initialization (RAII)

- `try`–`catch` párokat láttunk, de hol a finally?
  - A C++ nem támogatja a finally blokkokat
- Az RAII (Resource acquisition is initialization) „elvet” használja
  - Az objektum birtokolja az erőforrást
    - Az élettartalma alatt használható, megszűnésekor „felszabadul”

```
struct raii { /* TODO masolas implementalasa*/
  size_t n;
  int *p;
  raii(size_t n) : n(n), p(new int[n]) {
    cout << n << " elemu tomb letrehozva" << endl;
  }
  ~raii() {
    cout << n << " elemu tomb torolve" << endl;
    delete[] p;
  }
};
int main() {
  try {
    raii t(10);                                  // 10 elemu tomb letrehozva
    cin >> t.p[0];
    cout << "beolvasva: " << t.p[0] << endl; // beolvasva: 42
    throw exception();
  }                                              // 10 elemu tomb torolve
  catch (const exception& e) {
    cout << "hibakezeles ..." << endl;         // hibakezeles ...
  }
  cout << "vege" << endl;                        // vege
}
```

„Minden” elkapása

- Elkaphatunk bármilyen kivételt
  - Ha nem akarunk a kivétellel foglalkozni
    - Úgysem tudnánk a hibát megfelelően kezelni

```
int main() {
  try {
    datum d = ervenyes_datum(2019, 2, 222);
    cout << d.ev << " " << d.honap << " " << d.nap << endl;
  } catch (...) {
    cout << "Valami hiba tortent ..."  << endl;
  }
}
```

- Van, amikor a `catch(...)` sem kapja el a kivételt
  - Amikor egynél több kezeletlen kivétel van
    - Az `std::terminate` kerül meghívásra
  - Ha a kivétel nincs megadva a függvény fejlécében (ha van ilyen megadva)
    - Akkor `std::unexpected()` hívódik, ami pedig abort-ot hív

Dupla kivétel

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

class Kivetel {
public:
  Kivetel() {
    cout << "Kivetel()" << endl;
  }
  Kivetel(const Kivetel&) {
    cout << "Kivetel(const Kivetel&)" << endl;
    throw exception();
  }
};

int main() {
  try {
    try {
      throw Kivetel();
    } catch (Kivetel k) {
      cout << "kivetel elkapva" << endl;
    }
  } catch (...) {
    cout << "hiba" << endl; // nem fog kiirodni
  }
}

// Futas eredmenye:
// kivetel()
// kivetel(const kivetel&)
// libc++abi: terminating with uncaught exception of type std::exception: std::exception
```

Függvény specifikációs lista (nem kell tudni)

- Megadhatjuk a függvény fejlécében, hogy milyen típusú kivételt dobhat
  - Ha más dobódik, `std::unexpected()` kerül meghívásra
- Ha egy függvény semmit sem dobhat: `throw()`
  - C++11-ben elavultnak nyilvánították, C++20-ban eltávolították
- Ha csak bizonyos kivételeket, akkor: `throw(type1, type2)`
  - C++11-ben elavultnak nyilvánították, C++17-ben eltávolították
- Sajnos ez nem fordításidejű ellenőrzés, hanem futásidejű
- Általános vélekedés: soha ne használjuk (?)

```
int oszt(int a, int b) throw(int, std::exception) {
  if (b == 0)
    throw std::exception();  // Kivetel dobasa
  return a / b;
}
```

Függvény specifikációs lista

- `noexcept`
  - Ha jelezni szeretnénk, hogy a függvény nem dobhat kivételt, akkor arra a `noexcept` módosító szolgál (C++11 óta)
- `noexcept(expression)`
  - Ha az expression értéke igaz, akkor a függvény nem dob kivételt

```
// az f() fuggveny nem dob kivetelt
void f() noexcept;

// az f() fuggveny dobhat kivetelt
void g() noexcept(false);

// fp egy olyan fuggvenyre mutat, ami dobhat kivetelt
void (*fp)() noexcept(false);
```