OOP vizsgakérdések kidolgozás

Vári Gergő

2025. november 21.

Tartalomjegyzék

1 Ismertesse a C++ nyelvben alkalmazott bővítéseket az egyszerű adattípusok terén, valamint a konzol ki/bemenet megvalósításait! 1
1.1 C++ bővítések az egyszerű adattípusok terén . . . 1
1.2 Konzol Ki- és Bemenet (I/O) megvalósítása . . . 2

2 Ismertesse a csak C++ nyelvben alkalmazható típuskonverziót, valamint a header fájlok használatánál alkalmazható egyszerűsítést! 3
2.1 C++ specifikus típuskonverziók (Casting) . . . 3
2.2 Header fájlok használatának egyszerűsítése . . . 3

3 Ismertesse a referencia típusú változók deklarációját, használatát, és a referencia adattagot tartalmazó osztályok konstruktorának megvalósítását! Írjon példát a referencia típus használatára függvényben! 4
3.1 Referencia változók (Alias) . . . 4
3.2 Referencia adattagok és a konstruktor . . . 4
3.3 Referencia használata függvényekben . . . 5

4 Ismertesse a C++ nyelvben a függvények alapértelmezett paraméterezésének lehetőségét, és ennek szabályait! 6
4.1 Fogalma és Célja . . . 6
4.2 Szintaxis . . . 6
4.3 Alapvető Szabályok . . . 6
4.4 Kódpélda a szabályokra . . . 7

5 Ismertesse a C++ nyelvben a függvények túlterhelésének lehetőségét és ennek szabályait! 8
5.1 Fogalma és Lényege . . . 8
5.2 A megkülönböztetés szabályai . . . 8
5.3 Szigorú korlátok és hibalehetőségek . . . 8
5.3.1 1. Visszatérési érték (Return Type) . . . 8
5.3.2 2. Kétértelműség (Ambiguity) . . . 9
5.4 Összefoglaló táblázat . . . 9

6 Ismertesse a C++ nyelvben a template-ek működését függvény és osztály definiálása során! Írjon példán template-tel deklarált függvényre és használatára! 10
6.1 Fogalma és Célja . . . 10
6.2 Működési Mechanizmus . . . 10
6.3 Függvény és Osztály definiálása . . . 10
6.3.1 1. Függvény Template . . . 10
6.3.2 2. Osztály Template . . . 10
6.4 Példa: Függvény template használata . . . 10

7 Ismertesse a C++ nyelv memóriafoglalás és felszabadítás operátorait dinamikus példányok létrehozására és megszüntetésére! Írjon példát egy n elemű, double típusú adatokat tartalmazó tömb létrehozására és megszüntetésére! 12
7.1 Az operátorok áttekintése . . . 12
7.2 Skalár vs. Tömbös változatok . . . 12
7.3 Példa: n elemű double tömb kezelése . . . 12

8 Ismertesse a C++ hibakezelésben használható try-catch blokk működését! 14
8.1 Alapfogalmak és Cél . . . 14
8.2 A három fő komponens . . . 14
8.3 Működési folyamat . . . 14
8.4 Példa: Nullával való osztás kezelése . . . 14
8.5 Fontos szabályok . . . 15

9 Ismertesse az „egységbe zárás” objektum-orientált elvet! 16
9.1 Fogalma és Lényege . . . 16
9.2 Megvalósítás C++ nyelven . . . 16
9.3 Az egységbe zárás előnyei . . . 16
9.4 Példa: Bankszámla . . . 16

10 Ismertesse az „adatrejtés” objektum-orientált elvet! 18
10.1 Fogalma és Lényege . . . 18
10.2 Megvalósítás C++ nyelven . . . 18
10.3 Miért fontos? (Előnyök) . . . 18
10.4 Példa: Ellenőrzött hozzáférés . . . 18

11 Ismertesse az „öröklődés” objektum-orientált elvet! 20
11.1 Fogalma és Lényege . . . 20
11.2 Terminológia . . . 20
11.3 A `protected` (Védett) hozzáférés szerepe . . . 20
11.4 Szintaxis és Példa . . . 20
11.5 Összegzés . . . 21

12 Ismertesse a „sokalakúság” objektum-orientált elvet! 22
12.1 Fogalma és Lényege . . . 22
12.2 Technikai megvalósítás C++-ban . . . 22
12.3 A virtuális destruktor fontossága . . . 22
12.4 Példa: Állathangok . . . 22

13 Ismertesse a „this” pointer alkalmazását a fordító és a felhasználó szemszögéből! 24
13.1 Fogalma . . . 24
13.2 1. A Fordító szemszögéből (Implementáció) . . . 24
13.3 2. A Felhasználó (Programozó) szemszögéből . . . 24
13.4 Kódpélda a felhasználási esetekre . . . 24

14 Ismertesse a „private”, „protected”, „public” módosítók működését az osztálytagok definiálásakor! 26
14.1 A hozzáférési szintek célja . . . 26
14.2 A három módosító részletezése . . . 26
14.3 Összehasonlító táblázat . . . 26
14.4 Demonstrációs példa . . . 26

15 Ismertesse a „const” és „mutable” módosítók működését az osztálytagok definiálásakor! 28
15.1 Áttekintés . . . 28
15.2 A const módosító . . . 28
15.3 A mutable módosító . . . 28
15.4 Példakód . . . 29

16 Ismertesse a statikus adattagok tulajdonságait, megadási és elérési módjait! 30
16.1 Alapvető tulajdonságok . . . 30
16.2 Deklaráció és Definíció (Megadás) . . . 30
16.3 Elérési módok . . . 30
16.4 Példakód . . . 30

17 Ismertesse a „barátság” elvét és típusait az osztályok definiálásánál! 32
17.1 A barátság (friend) elve . . . 32
17.2 A barátság típusai . . . 32
17.3 Példakód . . . 32

18 Ismertesse a konstruktor működését! Mely konstruktorokat biztosítja a fordító alapértelmezetten? 34
18.1 A konstruktor működése . . . 34
18.2 A fordító által automatikusan biztosított konstruktorok . . . 34
18.3 Példakód . . . 35

19 Ismertesse a konstruktor megadásának szabályait! Milyen esetekben kell felülírnuk a fordító által definiált konstruktorokat? 36
19.1 A konstruktor megadásának szabályai . . . 36
19.2 Mikor kell felülírni a fordító által generált konstruktorokat? . . . 36
19.3 Példakód . . . 37

20 Ismertesse az adattagok kezdeti értékmegadásának lehetőségeit! Ezek közül melyik az, amelyik referencia típusú adattagok esetén használható? 38
20.1 Az értékmegadás lehetőségei . . . 38
20.2 Referencia típusú adattagok kezelése . . . 38
20.3 Példakód . . . 38

21 Ismertesse példával az 1 paraméterrel rendelkező konstruktor egyszerűsített meghívási lehetőségét! Hogyan tudjuk ezt az egyszerűsítést letiltani? 40
21.1 Egyszerűsített meghívás (Implicit konverzió) . . . 40
21.2 Az egyszerűsítés letiltása (explicit) . . . 40
21.3 Példakód . . . 40

22 Ismertesse a másoló konstruktor megírásának szükségességét okozó szituációt! Honnan tudjuk eldönteni, hogy a fordító a másoló konstruktort, vagy az „=” operátort használja? 42
22.1 Mikor szükséges saját másoló konstruktort írni? . . . 42
22.2 Másoló konstruktor vs. Értékadó operátor . . . 42
22.3 Példakód . . . 42

23 Ismertesse a destruktor definícióját, a destruktor készítés szabályait! Mit mondhatunk a destruktor kézi meghívásáról? 44
23.1 Definíció és Szerep . . . 44
23.2 A készítés szabályai . . . 44
23.3 A destruktor kézi meghívásáról . . . 44
23.4 Példakód . . . 45

24 Ismertesse a névterek definiálásának szükségességét a C++ programokban! Melyik operátorral hivatkozhatunk egy adott névtérben található osztályra? 46
24.1 A névterek (Namespaces) szükségessége . . . 46
24.2 Hivatkozás az elemekre . . . 46
24.3 Példakód . . . 46

25 Ismertesse az osztálypéldányokon végzett műveletek definiálási lehetőségeit! Mely műveleteket nem lehet átdefiniálni? 48
25.1 Az operátor-túlterhelés (Operator Overloading) lehetőségei 48
25.2 Nem átdefiniálható operátorok 48
25.3 Példakód 48

26 Ismertesse az osztályok kétoperandusú műveleteinek átdefiniálási lehetőségeit! Írjon példákat minden egyes lehetőséghez! 50
26.1 Áttekintés 50
26.2 1. Lehetőség: Tagfüggvényként (Member Function) 50
26.3 2. Lehetőség: Globális (Barát) függvényként (Global/Friend Function) 50
26.4 Példakód 50

27 Ismertesse az osztályok egyoperandusú műveleteinek átdefiniálási lehetőségeit! Írjon példákat minden egyes lehetőséghez! 52
27.1 Áttekintés 52
27.2 1. Lehetőség: Tagfüggvényként (Member Function) 52
27.3 2. Lehetőség: Globális (Barát) függvényként (Global Function) 52
27.4 Speciális eset: Prefix vs. Postfix (++ és –) 52
27.5 Példakód 52

28 Ismertesse a kommutatív műveletek átdefiniálási lehetőségét! Miért nem tudjuk a tagfüggvényes módszert alkalmazni? 54
28.1 A probléma: Miért nem jó a tagfüggvény? 54
28.2 A megoldás: Globális (Barát) függvény 54
28.3 Példakód 54

29 Ismertesse a „()” operátor túlterhelési lehetőségeit! 56
29.1 Áttekintés és Tulajdonságok 56
29.2 Gyakori felhasználási területek 56
29.3 Példakód 56

30 Ismertesse az „=” operátor túlterhelésének szintaktikáját és a szituációt, amelyben a fordító által biztosított operátor nem működik megfelelően! 58
30.1 Szintaktikai szabályok 58
30.2 A fordító által biztosított operátor problémája 58
30.3 Megvalósítási minta (Idiómák) 58
30.4 Példakód 59

31 Ismertesse a „new” és „delete” operátorok túlterhelésének szabályait! 60
31.1 Alapvető szintaxis és szignatúrák 60
31.2 Osztályszintű szabályok (Class-specific) 60
31.3 Globális szabályok (Global scope) 61
31.4 Kivételkezelés és konvenciók 61
31.5 Placement New szabályai 61

32 Ismertesse az I/O operátorok túlterhelésének szabályait! Írjon példát osztálypéldány kiíratásához! 62
32.1 Alapvető szabályok és elhelyezkedés 62
32.2 A függvények szignatúrája 62
32.3 A „friend” mechanizmus szerepe 62
32.4 Példa: Komplex szám osztály kiíratása 63

33 Ismertesse az „std” névtér „string” osztályát! Adja meg (működés magyarázatával) gyakran használt operátorait és metódusait! 64
33.1 Általános jellemzők . . . 64
33.2 Gyakran használt operátorok . . . 64
33.3 Fontosabb tagfüggvények (metódusok) . . . 64
33.4 Példa a használatra . . . 65

34 Ismertesse a string-numerikus adat közti konverzióra használt osztályt! 66
34.1 Az osztály jellemzői és felépítése . . . 66
34.2 Fő metódusok és kezelés . . . 66
34.3 Konverziós irányok . . . 66
34.4 Mintapélda . . . 66

35 Ismertesse a fájlok kezelésére használt osztályt, gyakran használt metódusait és operátorait! 68
35.1 Az alapvető osztályok . . . 68
35.2 Megnyitás és fájlmódok . . . 68
35.3 Fontos tagfüggvények (Metódusok) . . . 68
35.4 Operátorok . . . 68
35.5 Példa: Írás és olvasás . . . 69

36 Ismertesse példával a „kompozíció” elvet osztályok egymásba ágyazására! 70
36.1 A kompozíció elvei . . . 70
36.2 Megvalósítás C++ nyelven . . . 70
36.3 Példa: Számítógép és Processzor . . . 70

37 Ismertesse az „aggregáció” elvet osztályok egymásba ágyazására! 72
37.1 Az aggregáció jellemzői . . . 72
37.2 Megvalósítás C++ nyelven . . . 72
37.3 Példa: Autó és Sofőr . . . 72

38 Ismertesse az „öröklődés” elvet osztályok egymásba ágyazására! Mi az öröklődés szintaktikája a C++-ban? 74
38.1 Az elv és a kapcsolat típusa . . . 74
38.2 Szintaktika C++ nyelven . . . 74
38.3 Származtatási módok (Láthatóság) . . . 74
38.4 Konstruktorok és Destruktorok sorrendje . . . 75
38.5 Példa . . . 75

39 Csoportosítsa az osztályban található elemeket öröklődési szempontból: mely elemek öröklődnek, és mely elemek nem öröklődnek? 76
39.1 Örökölt elemek . . . 76
39.2 Nem örökölt elemek . . . 76
39.3 Összefoglaló táblázat és példa . . . 77

40 Ismertesse öröklődés során a leszármazottban található konstruktor paraméterezésének és hívásának szabályait, tekintettel az ősben levő privát adattagokra! 78
40.1 A probléma: Privát adattagok elérése . . . 78
40.2 Konstruktor hívási szabályok . . . 78
40.3 Kötelezőség és Default konstruktor . . . 78
40.4 Példa . . . 78

41 Ismertesse az ősben található osztálytagok elérésének módosítását private és protected öröklődés során! 80
41.1 Az alapvető mechanizmus . . . 80
41.2 Protected (Védett) öröklődés . . . 80
41.3 Private (Privát) öröklődés . . . 80
41.4 Összefoglaló táblázat . . . 81
41.5 Láthatóság visszaállítása (Using declaration) . . . 81
41.6 Példa a korlátozásokra . . . 81

42 Ismertesse az osztálytagok elérését ősosztály típusú pointerrel! Mi a „korai kötés" működése és problémája? 83
42.1 Az ősosztály típusú mutató viselkedése . . . 83
42.2 A korai kötés (Early / Static Binding) . . . 83
42.3 A probléma: A polimorfizmus hiánya . . . 83
42.4 Példa a hibás működésre . . . 83

43 Ismertesse a C++-ban található „többszörös öröklődés" elvet! Az ismertetést ábrával és program-részlettel illusztrálja! 85
43.1 Alapvető szabályok . . . 85
43.2 Strukturális Ábra . . . 85
43.3 Problémák és megoldások . . . 85
43.4 Példa program . . . 85

44 Ismertesse példával a „virtuális metódus" elv működését! Mit tartalmaz a VMT (vftable) táblázat? A leszármazottban is ugyanazt a szintaktikát kell használni a virtuális metódus felülírásakor? 87
44.1 A működési elv: Késői kötés (Late Binding) . . . 87
44.2 A VMT (Virtual Method Table) felépítése . . . 87
44.3 Szintaktika a leszármazottban . . . 87
44.4 Példa . . . 88

45 Ismertesse ábrával a „közvetlen bázisosztály" és a „közvetett bázisosztály" fogalmakat! 89
45.1 Fogalmak definíciója . . . 89
45.2 Strukturális Ábra . . . 89
45.3 Programrészlet és Szintaktika . . . 89
45.4 Fontos szabályok . . . 90

46 Ismertesse ábrával a virtuális öröklődés szükségességét előidéző szituációt! 91
46.1 A probléma leírása . . . 91
46.2 Strukturális Ábra (A Rombusz) . . . 91
46.3 Megoldás: Virtuális öröklődés . . . 91
46.4 Példa a hibás és javított esetre . . . 91

47 Ismertesse a tisztán virtuális metódus készítésének szintaktikáját! Hogyan nevezzük a legalább 1 tisztán virtuális metódust tartalmazó osztályt? Milyen szabályok vonatkoznak erre az osztályra? 93
47.1 Szintaktika . . . 93
47.2 Elnevezés . . . 93
47.3 Az absztrakt osztályra vonatkozó szabályok . . . 93

48 Ismertesse az „overload" és „override" elvek közti különbséget, amennyiben ős és leszármazottban történő előfordulásról van szó! 94
48.1 1. Overload (Túlterhelés) . . . 94
48.2 2. Override (Felüldefiniálás) . . . 94
48.3 Összehasonlító táblázat . . . 94

48.4 Példa a két esetre . . . 94

49 Ismertesse a „static_cast” és „dynamic_cast” kulcsszavak működését! Hol fordulhat elő hibásan interpretált memória-terület? 96
49.1 static_cast (Fordítási idejű konverzió) . . . 96
49.2 dynamic_cast (Futásidejű konverzió) . . . 96
49.3 Hibásan interpretált memória-terület (Veszélyforrás) . . . 96
49.4 Példa a működésre és a hibára . . . 97

50 Ismertesse egy előre megírt programrendszer (netről letöltött, vagy eszközzel kapott SDK) használatának lépéseit C++-ban! 98
50.1 1. Az állományok előkészítése . . . 98
50.2 2. Fordítási beállítások (Compiler Settings) . . . 98
50.3 3. Szerkesztési beállítások (Linker Settings) . . . 98
50.4 4. Futásidejű feltételek (Runtime) . . . 98
50.5 Összefoglaló példa (CMake stílusban) . . . 99

51 Ismertesse a vector STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)! 100
51.1 Memória modell . . . 100
51.2 Bővíthetőség és Kapacitás . . . 100
51.3 Bejárás és Elemek elérése . . . 100
51.4 Példa a működésre . . . 101

52 Ismertesse a deque STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)! 102
52.1 Memória modell (Szegmentált felépítés) . . . 102
52.2 Bővíthetőség . . . 102
52.3 Bejárás és Elérés . . . 102
52.4 Példa a használatra . . . 102

53 Ismertesse a list STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)! 104
53.1 Memória modell (Láncolt szerkezet) . . . 104
53.2 Bővíthetőség és Módosítás . . . 104
53.3 Bejárás és Elérés . . . 104
53.4 Példa a használatra . . . 105

54 Ismertesse a set, multiset STL tárolók tulajdonságait (memória modell, bejárás, bővíthetőség)! 106
54.1 Memória modell (Kiegyensúlyozott fa) . . . 106
54.2 Bővíthetőség és Teljesítmény . . . 106
54.3 Bejárás és Elérés . . . 106
54.4 Példa a működésre . . . 107

55 Ismertesse a map, multimap STL tárolók tulajdonságait (memória modell, bejárás, bővíthetőség)! 108
55.1 Memória modell (Fa szerkezet) . . . 108
55.2 Bővíthetőség és Műveletek . . . 108
55.3 Bejárás . . . 108
55.4 Példa a használatra . . . 109

56 Ismertesse az STL tároló adaptereket és működésüket! Melyik mire használható? 110
56.1 1. std::stack (Verem) . . . 110
56.2 2. std::queue (Sor) . . . 110
56.3 3. std::priority_queue (Prioritásos sor) . . . 111
56.4 Példa a használatra . . . 111

57 Ismertesse az STL iterátorok működését és feladatát egy lista STL tároló esetén! 113
57.1 Az iterátor feladata és fogalma . . . 113
57.2 Működés std::list esetén (Láncolt lista) . . . 113
57.3 Az iterátor kategóriája: Kétirányú (Bidirectional) . . . 113
57.4 Iterátor érvényesség (Iterator Validity) . . . 113
57.5 Példa a használatra . . . 114

58 Ismertesse az STL tárolókon végrehajtható algoritmusok működését és testre szabási lehetőségeiket! 115
58.1 Működési elv: Az iterátorok szerepe . . . 115
58.2 Algoritmus típusok . . . 115
58.3 Testre szabás (Customization) . . . 115
58.4 Példa: Rendezés és Keresés testre szabása . . . 116

59 Ismertesse a „nyelvi változó", „karakterisztikus függvény" és a „tagsági függvény" fogalmakat Zadeh szerint! 117
59.1 1. Karakterisztikus függvény (Hagyományos halmazok) . . . 117
59.2 2. Tagsági függvény (Fuzzy halmazok) . . . 117
59.3 3. Nyelvi változó (Linguistic Variable) . . . 117
59.4 Összefüggés a fogalmak között . . . 118

60 Ismertesse példával azt a szituációt, amikor egy fuzzy partíció lefedi az alaphalmazt! Adja meg a szöveges definíciót is! 119
60.1 Szöveges és Formális Definíció . . . 119
60.2 Gyakorlati Példa: Víz hőmérséklet szabályozás . . . 119

61 Ismertesse ábrával a „mag", „tartó", „α vágat", „szigorú α vágat" és „magasság" fogalmakat a fuzzy halmazok esetén! 120
61.1 Szemléltető Ábra . . . 120
61.2 Definíciók . . . 120

62 Ismertesse a Zadeh szerinti „s-norma", „t-norma" és komplemens képzését fuzzy halmazoknál! 122
62.1 1. Fuzzy Komplemens (Tagadás) . . . 122
62.2 2. T-norma (Fuzzy Metszet / ÉS) . . . 122
62.3 3. S-norma (Fuzzy Unió / VAGY) . . . 122
62.4 Összefüggés: De Morgan azonosságok . . . 123
62.5 Implementációs példa . . . 123

63 Ismertesse, hogy mikor alkalmazható egy t-norma, s-norma, komplemens definicióit tartalmazó szabályrendszer fuzzy halmazműveletekhez! Mit alkotnak ilyenkor a szabályrendszer elemei? 124
63.1 Alkalmazhatóság feltétele: A De Morgan Hármas . . . 124
63.2 Mit alkotnak az elemek? . . . 124

64 Ismertesse a fuzzy rendszerek általános blokkvázlatát! 126
64.1 A rendszer felépítése (Strukturális ábra) . . . 126
64.2 1. Fuzzifikáló egység (Fuzzification Interface) . . . 126
64.3 2. Tudásbázis (Knowledge Base) . . . 126
64.4 3. Következtető motor (Inference Engine) . . . 126
64.5 4. Defuzzifikáló egység (Defuzzification Interface) . . . 127

65 Ismertesse példával a fuzzy következtető módszer működését! 128
65.1 A példa paraméterei . . . 128
65.2 1. Lépés: Fuzzifikálás (Fuzzification) . . . 128
65.3 2. Lépés: Kiértékelés (Inference / Implication) . . . 128
65.4 3. Lépés: Aggregáció (Aggregation) . . . 128
65.5 4. Lépés: Defuzzifikálás (Defuzzification) . . . 129
65.6 Programkód (Szimuláció) . . . 129

66 Ismertessen defuzzifikációs módszereket! 130
66.1 1. Súlypont módszer (Center of Gravity - COG) . . . 130
66.2 2. Maximum középérték módszer (Mean of Maxima - MOM) . . . 130
66.3 3. Egyéb maximum-alapú módszerek . . . 130
66.4 4. Összegsúlypont módszer (Center of Sums - COS) . . . 131
66.5 Összehasonlító példa kódban . . . 131

67 Ismertesse az aggregációs operátorok definícióját, és 5 axiómáját! 132
67.1 Definíció . . . 132
67.2 Az 5 alapvető axióma (Tulajdonság) . . . 132
67.3 Osztályozás az idempotencia alapján . . . 132

68 Ismertesse az általános hatványközép operátort, és a paraméter speciális eseteiben elnevezett értékeit! 133
68.1 Definíció . . . 133
68.2 Speciális esetek (Nevezetes közepek) . . . 133
68.3 Tulajdonságok . . . 134

69 Ismertesse az OWA operátort! 135
69.1 Definíció . . . 135
69.2 A legfontosabb különbség a Súlyozott Átlaghoz képest . . . 135
69.3 Speciális esetek (A súlyvektor függvényében) . . . 135
69.4 Jellemzők: Orness és Diszperzió . . . 135
69.5 Számítási Példa . . . 136

1 Ismertesse a C++ nyelvben alkalmazott bővítéseket az egyszerű adattípusok terén, valamint a konzol ki/bemenet megvalósításait!

1.1 C++ bővítések az egyszerű adattípusok terén

A C++ a C nyelv típusrendszerére épül, de szigorúbb típusellenőrzést vezet be, és számos új elemmel bővíti azt a biztonságosabb és kényelmesebb programozás érdekében.

- **Logikai típus (`bool`):**
  - A C-vel ellentétben (ahol az egész számok 0/nem-0 értéke jelentette a logikát) a C++ bevezeti az önálló `bool` típust.
  - Lehetséges értékei: `true` (igaz) és `false` (hamis).
  - Memóriaigénye: implementációfüggő, de általában 1 bájt.
- **Referencia típus (`&`):**
  - Ez egy már létező változó "álneve" (alias).
  - **Jellemzői:** Deklaráláskor azonnal inicializálni kell, és később nem állítható át más változóra.
  - **Előnye:** Lehetővé teszi a cím szerinti paraméterátadást a mutatók (pointerek) bonyolult szintaxisa nélkül (pl. `void fv(int &x)`).
- **Inicializálási módok bővülése:**
  - **Konstruktor szintaxis:** `int a(5);` – úgy néz ki, mint egy függvényhívás.
  - **Egységes inicializálás (C++11):** `int a{5};` – ez a legbiztonságosabb forma (megakadályozza a szűkítő konverziót).
- **Típuslevezetés (`auto`):**
  - A fordító a kezdőérték alapján automatikusan határozza meg a változó típusát.
  - Példa: `auto x = 5.5;` (a típus `double` lesz).
  - Kötelező inicializálni a használatakor.
- **`nullptr`:**
  - A típusbiztonság érdekében bevezették a `nullptr` kulcsszót a NULL makró (ami sima 0) helyett, így elkerülhetők a pointerek és egészek keveredéséből adódó hibák.

1.2 Konzol Ki- és Bemenet (I/O) megvalósítása

A C++ az objektumorientált `iostream` könyvtárat használja a `cstdio` (printf/scanf) helyett/mellett. Az I/O műveletek adatfolyamokként (streamek) valósulnak meg.

- **Könyvtár és Névtér:**
  - Header fájl: `#include <iostream>`
  - Minden elem az `std` névtérben található (pl. `std::cout`).
- **Alapvető objektumok (Streamek):**
  - `cin`: Szabványos bemenet (billentyűzet).
  - `cout`: Szabványos kimenet (képernyő/konzol).
  - `cerr`: Szabványos hibakimenet (pufferelés nélküli).
  - `clog`: Szabványos naplózó kimenet (pufferelt).
- **Operátorok:**
  - **Beszúró operátor («):** Adatot küld a kimeneti folyamra. Láncolható.
    Példa: `cout « "Ertek:  " « x;`
  - **Kiemelő operátor (»):** Adatot olvas a bemeneti folyamról egy változóba. Automatikusan kezeli a típusokat és figyelmen kívül hagyja a szóközöket (whitespace).
    Példa: `cin » x;`
- **Manipulátorok (Formázás):**
  - A kimenet formázását végző speciális elemek.
  - `std::endl`: Új sor beszúrása és a puffer ürítése (flush).
  - További formázások az `<iomanip>` headerben: `setw()` (mezőszélesség), `setprecision()` (tizedesjegyek).
- **Előnyök a C (printf) megoldással szemben:**
  - **Típusbiztonság:** Nem kellenek formátumkódok (pl. `%d`), a fordító felismeri a típusokat.
  - **Bővíthetőség:** Saját osztályokra/típusokra is túlterhelhetők a « és » operátorok.

2 Ismertesse a csak C++ nyelvben alkalmazható típuskonverziót, valamint a header fájlok használatánál alkalmazható egyszerűsítést!

2.1 C++ specifikus típuskonverziók (Casting)

A C-stílusú `(tipus)ertek` konverzió helyett a C++ négy speciális operátort vezetett be. Ezek célja a típusbiztonság növelése és a szándék egyértelmű jelzése a kódban (könnyebb kereshetőség).

- `static_cast<T>(kif)` – A "logikus" konverzió
  - **Felhasználás:** Kompatibilis típusok között (pl. `int` → `float`, `enum` → `int`, pointer upcast).
  - **Ellenőrzés:** Fordítási időben (Compile-time).
  - **Biztonság:** Ha a típusok logikailag nem konvertálhatók, a fordító hibát jelez.
- `dynamic_cast<T>(kif)` – Az "objektumorientált" konverzió
  - **Felhasználás:** Polimorf osztályoknál (van `virtual` függvénye) öröklődési fában lefelé (downcast).
  - **Ellenőrzés:** Futási időben (Runtime check).
  - **Biztonság:** Ellenőrzi, hogy az objektum ténylegesen az-e, aminek hisszük.
    - Pointer esetén: sikertelenségnél `nullptr`-t ad.
    - Referencia esetén: sikertelenségnél `std::bad_cast` kivételt dob.
- `const_cast<T>(kif)` – A "szabálytalanító"
  - **Felhasználás:** A `const` vagy `volatile` minősítő levétele.
  - **Cél:** Főleg régebbi (legacy) kódok illesztéséhez, ahol egy függvény nem módosít, de lemaradt a paraméteréből a `const`.
- `reinterpret_cast<T>(kif)` – A "brutális" átértelmezés
  - **Felhasználás:** Alacsony szintű bit-manipuláció (pl. pointer konvertálása `int`-té, vagy két teljesen független pointer típus között).
  - **Jellemző:** Nincs ellenőrzés, a biteket változatlanul hagyja, csak másképp értelmezi. Veszélyes és nem hordozható (platformfüggő).

2.2 Header fájlok használatának egyszerűsítése

A nagy projektek fordítási idejének csökkentése és a körkörös függőségek (circular dependency) elkerüléséhez.

- **Include Guard modernizálása (`#pragma once`)**
  - **Probléma:** Egy header fájl többszörös beillesztése fordítási hibát okoz.
  - **Hagyományos megoldás:** `#ifndef LABEL`, `#define LABEL`, `#endif`.
  - **Egyszerűsítés:** A fájl legelső sorába írt `#pragma once`.
  - **Előny:** Nem kell egyedi makróneveket kitalálni, rövidebb, tisztább a kód, a fordító optimalizálhatja.
- **Elődeklaráció (Forward Declaration)**
  - **Módszer:** A header fájlban nem `#include "Osztaly.h"`-t használunk, hanem csak kiírjuk: `class Osztaly;`.
  - **Feltétele:** Csak akkor működik, ha az adott headerben csak pointert (`Osztaly*`) vagy referenciát (`Osztaly&`) használunk az adott típusból (nem példányosítjuk, nem érjük el a mezőit).
  - **Előny:**
    - Drasztikusan csökken a fordítási idő (kevesebb fájlt kell feldolgozni).
    - Megszünteti a körkörös hivatkozásokat (A include B, B include A).

3 Ismertesse a referencia típusú változók deklarációját, használatát, és a referencia adattagot tartalmazó osztályok konstruktorának megvalósítását! Írjon példát a referencia típus használatára függvényben!

3.1 Referencia változók (Alias)

A referencia egy már létező objektum vagy változó **másodlagos neve** (álneve). A C++ nyelvben a pointerek biztonságosabb és kényelmesebb alternatívájaként szolgál számos esetben.

- **Deklaráció:** A típus után tett `&` jellel.
- **Kötelező inicializálás:** Deklaráláskor azonnal hozzá kell kötni egy létező változóhoz.
- **Rögzítettség:** Az inicializálás után **nem irányítható át** más változóra.
- **Nincs NULL érték:** A referenciának mindig érvényes memóriaterületre kell mutatnia.

```
int x = 10;
int& ref = x;  // 'ref' mostantól 'x' álneve

ref = 20;      // Ez valójában 'x'-et módosítja 20-ra
// int& hibas; // HIBA: nincs inicializálva!
```

3.2 Referencia adattagok és a konstruktor

Ha egy osztály referencia típusú adattagot tartalmaz, speciális szabályok vonatkoznak a konstruktorra, mivel a referenciát létrehozáskor inicializálni kell.

- **Probléma:** A konstruktor törzsének futásakor az adattagok már létrejöttek (memória lefoglalva), így ott már késő lenne értéket adni a referenciának (az már értékadás lenne, nem inicializálás).
- **Megoldás:** Kötelező a **taginicializáló lista** (Member Initializer List) használata.

```
class AdatTarolo {
    int& referenciaAdat; // Referencia adattag

public:
    // Konstruktor
    AdatTarolo(int& kulsoValtozo)
        : referenciaAdat(kulsoValtozo) // Taginicializáló lista
    {
        // A törzsben (itt) már hiba lenne értéket adni neki először.
    }
};
```

3.3 Referencia használata függvényekben

A referenciák leggyakoribb felhasználása a függvényparamétereknél történik (*Pass-by-Reference*).

- **Hatékonyság:** Nem készül másolat az objektumról (nagy adatstruktúráknál kritikus).
- **Módosíthatóság:** A függvény képes megváltoztatni a hívó fél eredeti változóját (kimenő paraméter).
- **Konstans referencia:** Ha a másolást el akarjuk kerülni, de a módosítást tiltani szeretnénk: `const Type&`.

```
// Két szám cseréje (referenciák nélkül nem működne másolás miatt)
void csere(int& a, int& b) {
    int temp = a;
    a = b; // Az eredeti változót írja felül
    b = temp;
}

int main() {
    int x = 5, y = 10;
    csere(x, y);
    // Itt: x = 10, y = 5
    return 0;
}
```

4 Ismertesse a C++ nyelvben a függvények alapértelmezett paraméterezésének lehetőségét, és ennek szabályait!

4.1 Fogalma és Célja

- **Definíció:** Lehetőség arra, hogy a függvény paramétereinek előre megadott (default) értéket rendeljünk.
- **Működés:** Ha a függvényhívás során a hívó fél nem ad meg argumentumot az adott paraméterhez, a fordító automatikusan az alapértelmezett értéket illeszti be.
- **Haszna:**
  - Függvénytúlterhelés (function overloading) egyszerűsítése vagy kiváltása.
  - A kód olvashatóságának növelése.
  - Meglévő függvények bővítése új paraméterekkel a meglévő hívások "eltörése" nélkül.

4.2 Szintaxis

Az alapértelmezett értéket az értékadó operátorral (=) adjuk meg a paraméter típusát és nevét követően.

```
// Deklaráció (prototípus)
void ablakNyit(int szelesseg, int magassag, bool teljesKepernyo = false);

// Hívások
ablakNyit(800, 600);        // teljesKepernyo = false (alapértelmezett)
ablakNyit(1920, 1080, true); // teljesKepernyo = true (felülírt)
```

4.3 Alapvető Szabályok

Az alábbi szabályok ismerete elengedhetetlen a helyes használathoz:

1. **Jobbról-balra szabály (Right-to-left rule):** Ha egy paraméternek alapértelmezett értéket adunk, akkor az **összes** utána következő (tőle jobbra lévő) paraméternek is rendelkeznie kell alapértelmezett értékkel.
2. **Deklaráció vs. Definíció:** Az alapértelmezett értékeket általában a függvény **deklarációjában** (header fájlban, prototípusban) adjuk meg.
   - Ha a definíció (implementáció) külön van, ott **TILOS** megismételni az alapértékeket (fordítási hiba, még akkor is, ha ugyanazt az értéket írnánk be).
3. **Argumentumok elhagyása:** Argumentumokat csak a paraméterlista **végéről** hagyhatunk el. Nem lehet "lyukasan" paraméterezni (pl. az elsőt és harmadikat megadjuk, de a középsőt az alapértékre bízzuk).

4.4 Kódpélda a szabályokra

```cpp
// HELYES: Jobbról balra haladva mindenki kapott értéket
void teszt(int a, int b = 5, int c = 10);

// HELYTELEN: 'b'-nek van default értéke, de a tőle jobbra lévő 'c'-nek nincs
// void hibas(int a, int b = 5, int c); // -> Fordítási hiba!

// Implementáció: Itt már NEM szerepelnek az egyenlőségjelek
void teszt(int a, int b, int c) {
    // ... kód ...
}

int main() {
    teszt(1);       // a=1, b=5, c=10
    teszt(1, 2);    // a=1, b=2, c=10
    teszt(1, 2, 3); // a=1, b=2, c=3

    // teszt(1, , 3); // -> Fordítási hiba! (nem lehet kihagyni a közepét)
    return 0;
}
```

5 Ismertesse a C++ nyelvben a függvények túlterhelésének lehetőségét és ennek szabályait!

5.1 Fogalma és Lényege

- **Definíció:** A függvénytúlterhelés (function overloading) lehetővé teszi, hogy több függvény **azonos névvel**, de **eltérő paraméterlistával** (szignatúrával) létezzen egy hatókörön (scope) belül.
- **Polimorfizmus:** Ez a statikus (fordítási idejű) polimorfizmus egyik formája.
- **Működés:** A fordító a híváskor átadott argumentumok típusa és száma alapján választja ki a megfelelő függvényváltozatot (Resolution).

5.2 A megkülönböztetés szabályai

A C++ fordító a szignatúra alapján különbözteti meg a függvényeket. A túlterhelés akkor érvényes, ha az alábbiak közül legalább egy teljesül:

1. **Paraméterek száma:** Eltérő darabszámú argumentum.
2. **Paraméterek típusa:** A paraméterek típusai különböznek.
3. **Paraméterek sorrendje:** Különböző típusú paraméterek eltérő sorrendben szerepelnek.

```
void print(int a);              // Alap
void print(double a);           // Eltérő típus
void print(int a, int b);       // Eltérő darabszám
void print(char* c);            // Eltérő típus
```

5.3 Szigorú korlátok és hibalehetőségek

5.3.1 1. Visszatérési érték (Return Type)

Nem lehet függvényt túlterhelni kizárólag a visszatérési érték típusa alapján.

- *Ok:* A függvényhíváskor nem mindig használjuk fel a visszatérési értéket, így a fordító nem tudná eldönteni, melyik verziót futtassa.

```
int szamol(int x) { return x; }
double szamol(int x) { return x * 1.5; }
// -> FORDÍTÁSI HIBA: Újradefiniálás, a paraméterlista azonos!
```

5.3.2 2. Kétértelműség (Ambiguity)

Olyan helyzet, amikor a fordító több lehetséges függvényt is talál, és nem tud dönteni (pl. automatikus típuskonverziók vagy default paraméterek miatt).

```
void teszt(float f);
void teszt(double d);

// Hívás:
teszt(3.14);
// -> Sikeres: double literál, a másodikat hívja.

teszt(5);
// -> HIBA (Ambiguous call):
// Az int konvertálható float-tá és double-lé is.
// A fordító nem tud választani.
```

5.4 Összefoglaló táblázat

Megkülönböztető tényező	**Lehet túlterhelni?**
Paraméterek száma	IGEN
Paraméterek típusa	IGEN
Paraméterek sorrendje (eltérő típusnál)	IGEN
Csak visszatérési érték	**NEM**
Csak paraméter neve	**NEM**

6 Ismertesse a C++ nyelvben a template-ek működését függvény és osztály definiálása során! Írjon példán template-tel deklarált függvényre és használatára!

6.1 Fogalma és Célja

- **Generikus programozás:** A template-ek (sablonok) teszik lehetővé a C++-ban a típusfüggetlen kódírást.
- **Cél:** Olyan algoritmusok vagy adatszerkezetek írása, amelyeket egyszer definiálunk, de különféle adattípusokkal (pl. `int`, `double`, saját objektum) is működnek.
- **"Tervrajz":** A template nem egy kész függvény vagy osztály, hanem egy minta, amiből a fordító generálja le a tényleges kódot.

6.2 Működési Mechanizmus

- **Fordítási idő (Compile-time):** A template kiértékelése fordítási időben történik.
- **Példányosítás (Instantiation):** Amikor a sablont egy konkrét típussal használjuk, a fordító létrehoz (generál) belőle egy dedikált verziót az adott típusra.
- **Kulcsszavak:** A `template` kulcsszóval vezetjük be, utána csúcsos zárójelben adjuk meg a típusparamétereket (pl. `<typename T>` vagy `<class T>`). A kettő jelentése ebben a kontextusban megegyezik.

6.3 Függvény és Osztály definiálása

6.3.1 1. Függvény Template

- Olyan függvény, amelynek paraméterei vagy visszatérési értéke általános típusú.
- **Típuslevezetés (Type Deduction):** Híváskor a fordító általában automatikusan kitalálja a típust az átadott argumentumokból (nem kötelező kiírni, hogy `<int>`).

6.3.2 2. Osztály Template

- Olyan osztály, ahol az adattagok típusa típusparaméterként van megadva.
- **Használat:** Tipikus példák a tárolók (konténerek), pl. `std::vector` vagy `std::list`.
- **Explicit megadás:** Objektum létrehozásakor régebbi C++ szabványokban kötelező, újabbakban (C++17 óta) ajánlott megadni a típust (pl. `Osztaly<int> obj;`).

6.4 Példa: Függvény template használata

Az alábbi példa egy maximum-kiválasztó függvényt mutat be, amely bármilyen összehasonlítható típussal működik.

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

// --- Sablon definíciója ---
// T: helyettesítő típusparaméter
template <typename T>
T maximum(T a, T b) {
    if (a > b) {
        return a;
```

```cpp
    } else {
        return b;
    }
}

int main() {
    // 1. Használat egész számokkal (automatikus típuslevezetés: int)
    int x = 10, y = 20;
    std::cout << "Max int: " << maximum(x, y) << std::endl;

    // 2. Használat lebegőpontos számokkal (automatikus: double)
    double d1 = 5.5, d2 = 2.3;
    std::cout << "Max double: " << maximum(d1, d2) << std::endl;

    // 3. Használat explicit típusmegadással
    // Hasznos, ha a paraméterek típusa eltérne, de kényszeríteni akarjuk
    std::cout << "Explicit: " << maximum<double>(5, 6.7) << std::endl;

    return 0;
}
```

7 Ismertesse a C++ nyelv memóriafoglalás és felszabadítás operátorait dinamikus példányok létrehozására és megszüntetésére! Írjon példát egy n elemű, double típusú adatokat tartalmazó tömb létrehozására és megszüntetésére!

7.1 Az operátorok áttekintése

A C++ nyelvben a dinamikus memóriakezelés (Heap/Halom használata) dedikált operátorokkal történik, amelyek típusbiztosak és kezelik az objektumok életciklusát.

- **`new` operátor:**
  - Lefoglalja a megfelelő méretű memóriaterületet a Heap-en.
  - **Meghívja a konstruktort**, így az objektum azonnal inicializálva lesz.
  - Visszatér a lefoglalt területre mutató, típushelyes pointerrel.
  - Hiba esetén `std::bad_alloc` kivételt dob.
- **`delete` operátor:**
  - **Meghívja a destruktort**, így az objektum elvégezheti a takarítást (pl. fájlok lezárása).
  - Felszabadítja a memóriát és visszaadja az operációs rendszernek.

7.2 Skalár vs. Tömbös változatok

Szigorú szabály a megfelelő párok használata. A formák keverése definiálatlan viselkedéshez (memóriaszivárgás vagy összeomlás) vezet.

Használat	Foglalás (Allocation)	Felszabadítás (Deallocation)
Egyetlen objektum	`new Típus`	`delete ptr`
Tömb (Array)	`new Típus[méret]`	`delete[] ptr`

7.3 Példa: n elemű double tömb kezelése

Az alábbi kódrészlet bemutatja a tömbös szintaxis (`new[]` és `delete[]`) helyes alkalmazását.

```cpp
#include <iostream>

void dinamikusTombKezeles() {
    int n;
    std::cout << "Kerem a tomb meretet: ";
    std::cin >> n;

    // 1. LÉTREHOZÁS (Allokáció)
    // A 'new double[n]' lefoglalja a memóriát n darab double számára.
    // Visszatérési érték: pointer az első elemre (double*).
    double* tomb = new double[n];

    // Használat (példa értékadás)
    for (int i = 0; i < n; ++i) {
        tomb[i] = static_cast<double>(i) * 1.5;
    }

    // ... itt használjuk a tömböt ...

```

```
    // 2. MEGSZÜNTETÉS (Deallokáció)
    // FONTOS: Mivel tömböt foglaltunk (new[]),
    // kötelező a 'delete[]' (szögletes zárójeles) forma használata!
    delete[] tomb;

    // Biztonsági lépés: A pointer nullázása,
    // hogy ne mutasson felszabadított területre (dangling pointer).
    tomb = nullptr;
}
```

8 Ismertesse a C++ hibakezelésben használható try-catch blokk működését!

8.1 Alapfogalmak és Cél

- **Cél:** A futásidejű hibák (runtime errors) strukturált kezelése a program összeomlásának elkerülése érdekében.
- **Elv:** A normál üzleti logika és a hibakezelő kód szétválasztása.
- **Kivétel (Exception):** Egy objektum vagy primitív érték, amely a hiba fellépésekor információt hordoz a problémáról.

8.2 A három fő komponens

1. `try` **(próbablock):** Ide kerül az a kód, amely potenciálisan hibát okozhat (kivételt dobhat).
2. `throw` **(dobás):** Ha hiba lép fel, ezzel az utasítással "dobjuk el" a kivételt. Ekkor a normál futás megszakad.
3. `catch` **(elfogó blokk):** Ez a blokk kapja el a dobott kivételt. Itt történik a hiba elhárítása, naplózása vagy a felhasználó értesítése.

8.3 Működési folyamat

- A program belép a `try` blokkba.
- Ha minden rendben, a `catch` blokkokat átugorja.
- Ha `throw` történik:
  1. A vezérlés azonnal kilép a `try` blokkból (a további utasítások nem futnak le).
  2. **Stack Unwinding (Verem visszabontása):** A `try` blokkban létrehozott lokális objektumok destruktorai lefutnak (memória felszabadul).
  3. A rendszer megkeresi a típusban illeszkedő `catch` ágat és átadja neki a vezérlést.

8.4 Példa: Nullával való osztás kezelése

```cpp
#include <iostream>

double osztas(double szamlalo, double nevezo) {
    if (nevezo == 0) {
        // Hiba jelzése egy kivétel dobásával
        // Itt most egy szöveget (const char*) dobunk
        throw "Hiba: Nullaval valo osztas!";
    }
    return szamlalo / nevezo;
}

int main() {
    try {
        // Védett kódblokk
        std::cout << "Eredmeny: " << osztas(10, 2) << std::endl; // OK
        std::cout << "Eredmeny: " << osztas(5, 0) << std::endl;  // Hiba!

        // Ez a sor már NEM fut le a hiba miatt:
        std::cout << "Ez nem jelenik meg." << std::endl;
```

```
20
21      } catch (const char* hibaUzenet) {
22          // Hiba elkapása és kezelése
23          std::cerr << "Kivetel elkapva: " << hibaUzenet << std::endl;
24      } catch (...) {
25          // "Joker" catch ág: minden egyéb típust elkap
26          std::cerr << "Ismeretlen hiba tortent!" << std::endl;
27      }
28
29      return 0;
30  }
```

8.5 Fontos szabályok

- **Típusillesztés:** A `catch` paraméterének típusának egyeznie kell a `throw` által dobott típusával (vagy annak ősosztályával).
- **Sorrend:** Több `catch` ág esetén a speciálisabb (leszármazott) kivételeket előbb kell elkapni, az általánosabbakat (ősosztály) később.
- **`catch(...)`**: A három ponttal jelölt catch blokk minden kivételt elkap, típustól függetlenül (általában utolsó ágként használjuk).

9 Ismertesse az „egységbe zárás” objektum-orientált elvet!

9.1 Fogalma és Lényege

- **Definíció:** Az egységbe zárás (Encapsulation) az adatoknak (tulajdonságok) és az azokon műveleteket végző függvényeknek (metódusok) egyetlen egységben, azaz **osztályban** történő összefogása.
- **Adatrejtés (Data Hiding):** Az objektum belső állapotát (adattagjait) elrejtjük a külvilág elől. Ezeket közvetlenül nem, csak ellenőrzött felületen keresztül lehet módosítani.
- **Black Box elv:** Az osztályt használó programozónak csak azt kell tudnia, *mit* csinál az objektum (publikus interfész), azt nem, hogy *hogyan* valósítja meg azt (implementáció).

9.2 Megvalósítás C++ nyelven

A hozzáférés szabályozása az úgynevezett hozzáférési módosítókkal (access modifiers) történik:

1. **`private` (Privát):**
   - Csak az osztály saját metódusai (és a barát osztályok) férnek hozzá.
   - **Szabály:** Az adattagokat szinte mindig ide rakjuk a védelem érdekében.
2. **`public` (Nyilvános):**
   - Bárhonnan elérhető.
   - **Szabály:** Azok a metódusok kerülnek ide, amelyeket a külvilág számára biztosítunk (interfész).
3. **`protected` (Védett):**
   - Kívülről nem látható (mint a private), de az öröklés során a leszármazott osztályok hozzáférhetnek.

9.3 Az egységbe zárás előnyei

- **Adatintegritás (Validáció):** A Setter metódusokban ellenőrizhetjük a bemenő adatokat (pl. egy életkor nem lehet negatív), így az objektum nem kerülhet érvénytelen állapotba.
- **Rugalmasság:** A belső implementáció (pl. az adattárolás módja) megváltoztatható anélkül, hogy a hívó oldali kódot módosítani kellene, amíg a publikus interfész változatlan marad.
- **Olvashatóság:** A kód logikailag összetartozó részei egy helyen vannak.

9.4 Példa: Bankszámla

A példában az egyenleg közvetlen módosítása le van tiltva, így garantálható, hogy negatív összeget nem lehet befizetni.

```
1  class Bankszamla {
2  private:
3      // ADATREJTÉS:
4      // A külvilág nem férhet hozzá közvetlenül, így nem tudja elrontani.
5      double egyenleg;
6
7  public:
8      // Konstruktor
9      Bankszamla() : egyenleg(0) {}
10
11     // Setter (Beállító) - Validációval
12     void befizet(double osszeg) {
```

```
        if (osszeg > 0) {
            egyenleg += osszeg;
        } else {
            // Hibás adat kezelése (pl. figyelmen kívül hagyás vagy hibaüzenet)
        }
    }

    // Getter (Lekérdező) - Csak olvasható hozzáférést ad
    double getEgyenleg() const {
        return egyenleg;
    }
};

int main() {
    Bankszamla szamla;

    // szamla.egyenleg = 10000; // -> FORDÍTÁSI HIBA: 'egyenleg' is private

    szamla.befizet(5000);       // HELYES: publikus metóduson át
    szamla.befizet(-200);       // ÉRVÉNYTELEN: a metódus kiszűri

    return 0;
}
```

10 Ismertesse az „adatrejtés” objektum-orientált elvet!

10.1 Fogalma és Lényege

- **Definíció:** Az adatrejtés (Data Hiding) az a technika, amellyel az objektum belső állapotát (adattagjait) elzárjuk a külvilág elől.
- **Cél:** Megakadályozni, hogy más programrészek véletlenül vagy szándékosan, ellenőrizetlenül módosítsák az objektum adatait.
- **Interfész és Implementáció szétválasztása:**
  - **Implementáció (Privát):** A belső működés részletei és az adatok tárolása. Ez kívülről láthatatlan.
  - **Interfész (Publikus):** Azok a metódusok (függvények), amelyeken keresztül kommunikálni lehet az objektummal.

10.2 Megvalósítás C++ nyelven

A hozzáférési szinteket (access specifiers) használjuk a láthatóság szabályozására:

1. `private` **(Privát):**
   - Az osztály alapértelmezett hozzáférése (class esetén).
   - Csak az osztály saját metódusai férnek hozzá.
   - **Ide helyezzük az adattagokat.**
2. `public` **(Nyilvános):**
   - Bárhonnan elérhető.
   - **Ide helyezzük a Getter/Setter metódusokat**, amelyek ellenőrzött hozzáférést biztosítanak a privát adatokhoz.

10.3 Miért fontos? (Előnyök)

- **Érvényesség (Validáció):** A Setter metódusokban megvizsgálhatjuk a kapott értéket. Ha érvénytelen (pl. negatív életkor), megakadályozhatjuk a beállítást.
- **Karbantarthatóság:** Ha megváltoztatjuk az adattárolás belső módját (pl. `int` helyett `long` vagy adatbázisból jön), a külvilágnak nem kell erről tudnia, amíg a publikus metódusok ugyanúgy hívhatók.
- **Csak olvashatóság:** Ha egy adathoz írunk Gettert, de Settert nem, akkor az adat kívülről "read-only" (csak olvasható) lesz.

10.4 Példa: Ellenőrzött hozzáférés

A példában a ‘Diak‘ osztály osztályzatait rejtjük el. Csak 1 és 5 közötti értéket engedünk beállítani.

```
1 class Diak {
2 private:
3     // REJTETT ADAT: Kívülről közvetlenül nem érhető el.
4     int jegy;
5
6 public:
7     // Konstruktor
8     Diak() : jegy(1) {}
9
```

```
    // SETTER (Beállító) - Validációval
    // Ez az "ajtó", amin keresztül módosítani lehet az adatot.
    void setJegy(int ujJegy) {
        if (ujJegy >= 1 && ujJegy <= 5) {
            jegy = ujJegy;
        } else {
            // Érvénytelen adat elutasítása
            // (Itt lehetne hibaüzenetet dobni vagy logolni)
        }
    }

    // GETTER (Lekérdező)
    // Ez biztosítja, hogy az adat olvasható legyen.
    int getJegy() const {
        return jegy;
    }
};

int main() {
    Diak d;

    // d.jegy = 6;  // -> FORDÍTÁSI HIBA: 'jegy' private!

    d.setJegy(4);   // Működik: érvényes adat
    d.setJegy(8);   // Nem történik semmi: érvénytelen adat, a védelem működik

    return 0;
}
```

11 Ismertesse az „öröklődés” objektum-orientált elvet!

11.1 Fogalma és Lényege

- **Definíció:** Az öröklődés (Inheritance) lehetővé teszi, hogy egy meglévő osztály (ős) tulajdonságait és viselkedését egy új osztály (utód) átvegye.
- **Cél:** A kód újrahasznosítása és a hierarchikus rendszerezés. Nem kell újra leírni a közös kódrészleteket.
- **"Is-a" kapcsolat:** Az öröklés "ez egy..." (is-a) kapcsolatot valósít meg. Például: Az *Autó* (utód) egy *Jármű* (ős).

11.2 Terminológia

- **Ősosztály (Base class / Super class):** Az az osztály, amelynek a tulajdonságait örökítjük.
- **Származtatott osztály (Derived class / Sub class):** Az új osztály, amely örökli az ős tulajdonságait, és általában újakkal egészíti ki azokat.

11.3 A `protected` (Védett) hozzáférés szerepe

Az öröklésnél megjelenik egy harmadik láthatósági szint:

- **`protected`:** A külvilág számára rejtett (mint a private), de a származtatott osztályok számára látható és módosítható.

11.4 Szintaxis és Példa

C++-ban az osztály neve után kettősponttal és a hozzáférési mód megadásával (általában `public`) jelöljük az öröklést.

```cpp
#include <iostream>

// ŐSOSZTÁLY (Szülő)
class Jarmu {
protected:
    int sebesseg; // A leszármazottak látják, a main() nem

public:
    Jarmu() : sebesseg(0) {}

    void indul() {
        std::cout << "A jarmu elindult." << std::endl;
    }
};

// SZÁRMAZTATOTT OSZTÁLY (Gyerek)
// Az Auto örökli a Jarmu minden tulajdonságát
class Auto : public Jarmu {
public:
    void gazadas() {
        // Hozzáférünk az ős 'protected' adattagjához
        sebesseg += 10;
        std::cout << "Sebesseg: " << sebesseg << " km/h" << std::endl;
    }

    // Saját, új funkció, ami az ősben nem volt
    void dudal() {
```

```
28          std::cout << "Tu-tu!" << std::endl;
29      }
30  };
31
32  int main() {
33      Auto kocsi;
34
35      // 1. Az ősosztály metódusát is tudja használni
36      kocsi.indul();
37
38      // 2. A saját metódusait is tudja használni
39      kocsi.gazadas();
40      kocsi.dudal();
41
42      return 0;
43  }
```

11.5 Összegzés

Az `Auto` osztálynak nem kellett újra definiálnia a `sebesseg` változót vagy az `indul()` függvényt, azokat "ingyen" megkapta a `Jarmu` osztálytól, így a kód rövidebb és átláthatóbb.

12 Ismertesse a „sokalakúság” objektum-orientált elvet!

12.1 Fogalma és Lényege

- **Definíció:** A sokalakúság (Polimorfizmus) lehetővé teszi, hogy különböző típusú objektumokat (amelyek egy közös ősosztályból származnak) egységes módon kezeljünk.
- **Egy interfész, több megvalósítás:** Ugyanaz a függvényhívás más és más viselkedést vált ki attól függően, hogy valójában milyen típusú objektum áll a háttérben.
- **Dinamikus kötés (Dynamic Binding):** A fordító nem fordítási időben, hanem futásidőben dönti el, melyik függvényt kell meghívni.

12.2 Technikai megvalósítás C++-ban

A futásidejű polimorfizmushoz három feltétel szükséges:

1. **Öröklődés:** Legyen egy közös ősosztály.
2. **Virtuális függvények (`virtual`):** Az ősosztályban a felüldefiniálandó metódusokat `virtual` kulcsszóval kell ellátni.
3. **Pointer vagy Referencia:** Az objektumokat az ősosztályra mutató pointeren vagy referencián keresztül kell meghívni.

12.3 A virtuális destruktor fontossága

Ha egy osztályban van virtuális függvény, a destruktornak is **virtuálisnak kell lennie** (`virtual ~Os()`).

- *Ok:* Ha az ősosztály pointerén keresztül törlünk (`delete`) egy objektumot, csak így garantálható, hogy a leszármazott osztály destruktora is lefusson és felszabadítsa a memóriát.

12.4 Példa: Állathangok

Ebben a példában a `main` függvény nem tudja (és nem is érdekli), hogy konkrétan milyen állatot kapott, csak azt tudja, hogy az egy `Allat`, és képes hangot adni.

```cpp
#include <iostream>

// ŐSOSZTÁLY
class Allat {
public:
    // Virtuális függvény: a leszármazottak felülírhatják
    virtual void hangotAd() {
        std::cout << "..." << std::endl;
    }

    // Virtuális destruktor: kötelező polimorfizmusnál!
    virtual ~Allat() {}
};

// LESZÁRMAZOTT 1
class Kutya : public Allat {
public:
    // 'override': jelzi, hogy szándékosan írjuk felül az őst
    void hangotAd() override {
        std::cout << "Vau!" << std::endl;
    }
```

```cpp
};

// LESZÁRMAZOTT 2
class Macska : public Allat {
public:
    void hangotAd() override {
        std::cout << "Miau!" << std::endl;
    }
};

int main() {
    // Polimorfizmus használata:
    // Ős típusú pointer mutat a leszármazottra
    Allat* bodri = new Kutya();
    Allat* cirmi = new Macska();

    // Ugyanaz a függvényhívás, eltérő viselkedés
    bodri->hangotAd(); // Kiírja: Vau!
    cirmi->hangotAd(); // Kiírja: Miau!

    // Takarítás
    delete bodri;
    delete cirmi;

    return 0;
}
```

13 Ismertesse a „this” pointer alkalmazását a fordító és a felhasználó szemszögéből!

13.1 Fogalma

- **Definíció:** A `this` egy speciális, konstans pointer, amely minden **nem statikus** tagfüggvényben (metódusban) automatikusan elérhető.
- **Hova mutat?** Mindig arra a konkrét objektumpéldányra mutat, amelyre az adott tagfüggvényt meghívták.
- **Típusa:** `OsztalyTipus* const` (maga a pointer nem változtatható meg, hogy más objektumra mutasson).

13.2 1. A Fordító szemszögéből (Implementáció)

A felhasználó számára a tagfüggvény hívása egyszerűnek tűnik (`obj.fv()`), de a háttérben a fordító átalakítja azt.

- **Rejtett paraméter:** A fordító minden tagfüggvényhez hozzáad egy láthatatlan, első paramétert: a `this` pointert.
- **Hívás átalakítása:** Amikor meghívunk egy metódust, a fordító átadja az objektum memóriacímét ennek a rejtett paraméternek.

Sematikus szemléltetés:

- **Amit mi írunk:** `obj.setAdat(5);`
- **Amit a fordító lát:** `setAdat(&obj, 5);`

13.3 2. A Felhasználó (Programozó) szemszögéből

Programozóként a `this`-t explicit módon kell használnunk bizonyos esetekben:

1. **Névütközés feloldása (Shadowing):** Ha a paraméter neve megegyezik az adattag nevével, a `this->` előtaggal hivatkozhatunk az osztály adattagjára.
2. **Metódusláncolás (Method Chaining):** Ha egy függvény a `*this` (az objektum dereferált értéke) referenciájával tér vissza, akkor több függvényhívás fűzhető egymás után (pl. `cout` vagy Builder minta).
3. **Ön-hivatkozás ellenőrzése:** Értékadó operátor (`operator=`) írásakor ellenőrizni kell, hogy nem önmagával egyenlővé tesszük-e az objektumot (`if (this == &other)`).

13.4 Kódpélda a felhasználási esetekre

```
#include <iostream>

class Szamolo {
private:
    int ertek;

public:
    Szamolo() : ertek(0) {}

    // 1. Névütközés kezelése
    // A paraméter neve 'ertek', ami eltakarja az adattagot.
    // A 'this->ertek' jelenti az osztály változóját.
```

```
13      void setErtek(int ertek) {
14          this->ertek = ertek;
15      }
16
17      // 2. Láncolhatóság
18      // A függvény visszatér az aktuális objektum referenciájával (*this).
19      Szamolo& hozzaad(int szam) {
20          this->ertek += szam;
21          return *this; // Visszaadjuk önmagunkat
22      }
23
24      void kiir() const {
25          std::cout << "Ertek: " << this->ertek << std::endl;
26      }
27  };
28
29  int main() {
30      Szamolo sz;
31
32      // Láncolt hívás:
33      // 1. beállítjuk 10-re
34      // 2. hozzáadunk 5-öt (eredmény: 15)
35      // 3. hozzáadunk 2-t (eredmény: 17)
36      sz.setErtek(10);
37      sz.hozzaad(5).hozzaad(2);
38
39      sz.kiir(); // 17
40      return 0;
41  }
```

14 Ismertesse a „private”, „protected”, „public” módosítók működését az osztálytagok definiálásakor!

14.1 A hozzáférési szintek célja

A C++ nyelvben a hozzáférési módosítók (access specifiers) szabályozzák az egységbe zárást (encapsulation). Azt határozzák meg, hogy az osztály adattagjaihoz és tagfüggvényeihez a kód mely részeiből lehet hozzáférni.

14.2 A három módosító részletezése

1. **`public` (Nyilvános):**
   - **Láthatóság:** Bárhonnan elérhető (az osztályon belülről, leszármazott osztályokból és a külvilágból, pl. a `main`-ből is).
   - **Használat:** Az osztály publikus interfésze (azok a függvények, amelyeket a felhasználónak szánunk).
2. **`protected` (Védett):**
   - **Láthatóság:** A külvilág számára rejtett, de az osztály saját metódusai és a **leszármazott osztályok** (gyerekosztályok) hozzáférhetnek.
   - **Használat:** Olyan segédváltozók vagy függvények, amelyek a belső működéshez kellenek, és szeretnénk megosztani az öröklési hierarchiában.
3. **`private` (Privát):**
   - **Láthatóság:** Kizárólag az adott osztály saját metódusai (és a `friend` osztályok) láthatják. Még a leszármazottak sem férnek hozzá!
   - **Használat:** Adattagok (belső állapot) védelme. Ez az alapértelmezett láthatóság `class` esetén.

14.3 Összehasonlító táblázat

Módosító	Saját osztály	Leszármazott	Külvilág (pl. main)
`public`	IGEN	IGEN	IGEN
`protected`	IGEN	IGEN	**NEM**
`private`	IGEN	**NEM**	**NEM**

14.4 Demonstrációs példa

```
class OsOsztaly {
public:
    int publikusAdat;    // Mindenki látja
protected:
    int vedettAdat;      // Csak a család (leszármazottak) látja
private:
    int privatAdat;      // Titok (csak ez az osztály látja)

public:
    void teszt() {
        privatAdat = 1;  // OK: Belül vagyunk
    }
};

class Leszarmazott : public OsOsztaly {
public:
    void hozzaferesTeszt() {
```

```
18          publikusAdat = 2; // OK
19          vedettAdat = 3;   // OK: Örököltük, láthatjuk
20
21          // privatAdat = 4; // -> HIBA: A gyerek sem láthatja a szülő privátját!
22      }
23  };
24
25  int main() {
26      OsOsztaly obj;
27
28      obj.publikusAdat = 10; // OK
29
30      // obj.vedettAdat = 20; // -> HIBA: Kívülről ez olyan, mintha private lenne
31      // obj.privatAdat = 30; // -> HIBA: Nem látható
32
33      return 0;
34  }
```

15 Ismertesse a „const” és „mutable” módosítók működését az osztálytagok definiálásakor!

15.1 Áttekintés

C++ objektumorientált programozásban a `const` és `mutable` kulcsszavak szabályozzák az objektumok állapotának módosíthatóságát, biztosítva a *const-correctness* elvét.

15.2 A const módosító

A `const` kulcsszó használata biztosítja, hogy az objektum állapota (bizonyos határok között) változatlan maradjon.

- **Const adattagok (adattag deklaráció):**
  - Az ilyen változók értéke az inicializálás után nem módosítható.
  - Értéket kizárólag a **konstruktor inicializáló listájában** kaphatnak.
  - Nem kaphatnak értéket a konstruktor törzsében (mivel ott már értékadás történne, nem inicializálás).
- **Const tagfüggvények (metódusok):**
  - A függvény deklarációjának végére írt `const` kulcsszó jelzi (pl. `int get() const;`).
  - **Garancia:** A függvény nem módosítja az objektum egyetlen (nem statikus) adattagját sem.
  - **This pointer:** A függvényen belül a `this` mutató típusa `const ClassName* const`-ra változik.
  - **Hívhatóság:** Konstans objektumon (`const ClassName obj`) kizárólag konstans tagfüggvények hívhatók meg.

15.3 A mutable módosító

A `mutable` (változékony) kulcsszó egy explicit kivétel a `const` szigorúsága alól.

- **Definíció:** Olyan adattagok előtt használjuk, amelyek akkor is módosíthatók maradnak, ha a tartalmazó objektum példánya `const`, vagy ha egy `const` tagfüggvényen belül vagyunk.
- **Bitonkénti vs. Logikai konstansság:**
  - A C++ fordító alapértelmezésben *bitonkénti konstansságot* ellenőriz (egyetlen bit sem változhat).
  - A `mutable` lehetővé teszi a *logikai konstansságot*: az objektum külső megfigyelő számára változatlannak tűnik, de belső adminisztrációs adatok változhatnak.
- **Gyakori felhasználási esetek:**
  - **Mutex-ek (szálbiztonság):** Egy `std::mutex` zárolása (`lock`) módosítja a mutex belső állapotát, de ez szükséges olvasási műveleteknél is.
  - **Cache-elés (Memoization):** Egy számításigényes `get()` függvény eredményének eltárolása az első híváskor, hogy később gyorsabban visszaadható legyen.
  - **Debug számlálók:** Hányszor hívtak meg egy metódust.

15.4 Példakód

Az alábbi példa bemutatja, hogyan módosítható egy `mutable` változó egy `const` függvényben a *Lazy Evaluation* (lusta kiértékelés) megvalósításához.

```
class ComplexCalculation {
private:
    int inputValue;

    // Cache változók: kívülről nem tartoznak az objektum
    // "lényegi" állapotához, ezért mutable-k.
    mutable int cachedResult;
    mutable bool isCalculated;

    // Szálbiztonság miatti mutex, mindig mutable
    mutable std::mutex mtx;

public:
    ComplexCalculation(int v)
        : inputValue(v), cachedResult(0), isCalculated(false) {}

    // A függvény const, tehát elvileg nem írhat adattagot
    int getResult() const {
        std::lock_guard<std::mutex> lock(mtx); // Mutex módosul!

        if (!isCalculated) {
            // Hosszú számítás szimulálása
            // A mutable miatt írhatjuk ezeket a tagokat:
            cachedResult = inputValue * inputValue;
            isCalculated = true;
        }
        return cachedResult;
    }

    // Setter: invalidálja a cache-t
    void setValue(int v) {
        std::lock_guard<std::mutex> lock(mtx);
        inputValue = v;
        isCalculated = false;
    }
};
```

16 Ismertesse a statikus adattagok tulajdonságait, megadási és elérési módjait!

16.1 Alapvető tulajdonságok

A statikus (`static`) adattagok olyan változók, amelyek nem egy konkrét objektumpéldányhoz, hanem magához az osztályhoz tartoznak.

- **Közös állapot:** Az osztály minden példánya ugyanazon az egyetlen változón osztozik. Ha az egyik objektum módosítja, az összes többi is az új értéket látja.
- **Példányfüggetlenség:** A statikus adattag akkor is létezik és elérhető, ha az osztályból még egyetlen példányt sem hoztunk létre.
- **Élettartam:** A program indulásakor (vagy az első használatkor) jönnek létre, és a program futásának végéig a memóriában maradnak (static storage duration).
- **Láthatóság:** Ugyanúgy vonatkoznak rájuk az elérési módosítók (`public`, `private`, `protected`), mint a normál adattagokra.

16.2 Deklaráció és Definíció (Megadás)

A statikus adattagok kezelése két lépésből áll (kivéve speciális eseteket): deklaráció az osztályon belül, és definíció (memória-allokáció) az osztályon kívül.

- **Deklaráció (Header fájlban):** Az osztály definícióján belül a `static` kulcsszóval jelöljük meg a változót. Ez csak a típusát és nevét közli a fordítóval.
- **Definíció (Forrásfájlban/.cpp):** Mivel a statikus tag nem tartozik példányhoz, a memóriát globális szinten kell lefoglalni neki, általában a .cpp fájlban. Itt már nem kell a `static` kulcsszó, de hivatkozni kell az osztályra (`Osztaly::`).
- **Kivételek (Inicializálás az osztályon belül):**
  - **const integral típusok:** Pl. `static const int MAX = 10;` megengedett az osztályon belül.
  - **inline static (C++17):** A `inline static` kulcsszóval ellátott tagok definíciója és inicializálása is történhet az osztályon belül, így nem kell külön .cpp definíció.

16.3 Elérési módok

A statikus adattagokhoz (ha publikusak) kétféleképpen férhetünk hozzá:

- **Osztálynév minősítéssel (Ajánlott):** Mivel az adat az osztályhoz tartozik, a `OsztalyNev::adattag` forma a legkifejezőbb. Ez egyértelműsíti, hogy nem példánytagról van szó.
- **Objektumpéldányon keresztül:** Használható az `objektum.adattag` vagy `pointer->adattag` forma is, de ez félrevezető lehet, mert azt sugallja, mintha az adat az objektumhoz tartozna.

16.4 Példakód

Az alábbi példa egy számlálót valósít meg, amely nyilvántartja, hány élő példány van az osztályból.

```
1 #include <iostream>
2
3 class Player {
4 private:
5     // 1. Deklaráció az osztályban
```

```
6       static int playerCount;
7
8   public:
9       Player() {
10          // Minden új példány növeli a közös számlálót
11          playerCount++;
12      }
13
14      ~Player() {
15          // Megszűnéskor csökkentjük
16          playerCount--;
17      }
18
19      // Statikus tagfüggvény a privát statikus adattag eléréséhez
20      static int getCount() {
21          return playerCount;
22      }
23  };
24
25  // 2. Definíció és inicializálás az osztályon kívül (globális scope)
26  // Itt foglalódik le a memória.
27  int Player::playerCount = 0;
28
29  int main() {
30      // Elérés példányosítás előtt (statikus metóduson keresztül)
31      std::cout << "Jatekosok: " << Player::getCount() << std::endl; // 0
32
33      Player p1;
34      Player p2;
35
36      // Elérés osztálynévvel (ajánlott)
37      std::cout << "Jatekosok: " << Player::getCount() << std::endl; // 2
38
39      {
40          Player p3;
41          // Elérés objektumon keresztül (működik, de nem idiomatikus)
42          // Bár p3.getCount()-ot írunk, ez statikus kötés
43          std::cout << "Jatekosok: " << p3.getCount() << std::endl; // 3
44      } // p3 megsemmisül
45
46      std::cout << "Jatekosok: " << Player::getCount() << std::endl; // 2
47      return 0;
48  }
```

17 Ismertesse a „barátság” elvét és típusait az osztályok definiálásánál!

17.1 A barátság (friend) elve

A C++ nyelvben a `friend` kulcsszó lehetővé teszi az adatrejtés (encapsulation) szabályozott megkerülését. A barátnak deklarált külső egységek hozzáférhetnek az osztály `private` és `protected` tagjaihoz is.

- **Kontrollált hozzáférés:** A barátságot mindig az az osztály *adja*, amelyiknek az adatait védeni kell (az osztályon belül kell deklarálni a barátot). A barátságot nem lehet „elvenni” vagy kívülről kikényszeríteni.
- **Nem szimmetrikus (Irányított):** Ha „A” osztály barátja „B”-nek, abból nem következik, hogy „B” is barátja „A”-nak.
- **Nem tranzitív:** Ha „A” barátja „B”-nek, és „B” barátja „C”-nek, abból nem következik, hogy „A” hozzáfér „C” privát adataihoz (a barát barátja nem a barátom).
- **Nem öröklődő:** A szülőosztály barátai nem válnak automatikusan a leszármazott osztály barátaivá.

17.2 A barátság típusai

Három fő módon definiálhatunk barátságot attól függően, hogy kinek adunk hozzáférést:

- **Barát függvény (Globális):**
  - Egy önálló, nem osztályhoz tartozó függvény kap hozzáférést.
  - **Tipikus használat:** Operátor kiterjesztés (overloading), például `operator«` (kimeneti adatfolyam) megvalósítása, ahol a bal oldali operandus (`std::ostream`) nem a mi osztályunk.
- **Barát osztály (Friend Class):**
  - Egy másik osztály teljes egészében barátnak van jelölve.
  - A barát osztály *minden* tagfüggvénye hozzáfér az eredeti osztály privát tagjaihoz.
  - **Tipikus használat:** Szorosan együttműködő osztályok (pl. `LinkedList` és `Node`, vagy `Manager` és `Worker`).
- **Barát tagfüggvény (Friend Member Function):**
  - Nem a teljes osztályt, csak egy másik osztály egyetlen konkrét metódusát tesszük baráttá.
  - Precízebb hozzáférést biztosít, de erősebb függőséget (include sorrend) igényel a definíciónál.

17.3 Példakód

Az alábbi példa bemutatja a barát osztály és a barát (globális) függvény használatát.

```
#include <iostream>

class Tarolo {
private:
    int titkosAdat;

public:
    Tarolo(int adat) : titkosAdat(adat) {}

    // 1. Barát osztály deklarációja
    // A Ellenor osztály minden metódusa láthatja a titkosAdat-ot
```

```
12      friend class Ellenor;
13
14      // 2. Barát globális függvény deklarációja
15      // Ez a függvény nem tagja az osztálynak!
16      friend void kiir(const Tarolo& t);
17  };
18
19  // --- Barát osztály definíciója ---
20  class Ellenor {
21  public:
22      void vizsgal(const Tarolo& t) {
23          // Hozzáférés a privát adattaghoz
24          if (t.titkosAdat > 10) {
25              std::cout << "Az adat nagy." << std::endl;
26          }
27      }
28
29      void nullaaz(Tarolo& t) {
30          // Módosítani is tudja
31          t.titkosAdat = 0;
32      }
33  };
34
35  // --- Barát globális függvény definíciója ---
36  void kiir(const Tarolo& t) {
37      // Itt is elérjük a privát tagot
38      std::cout << "Ertek: " << t.titkosAdat << std::endl;
39  }
40
41  int main() {
42      Tarolo doboz(42);
43
44      Ellenor inspektor;
45      inspektor.vizsgal(doboz); // OK
46
47      kiir(doboz); // OK
48
49      // doboz.titkosAdat; // HIBA: a main nem barát!
50      return 0;
51  }
```

18 Ismertesse a konstruktor működését! Mely konstruktorokat biztosítja a fordító alapértelmezetten?

18.1 A konstruktor működése

A konstruktor egy speciális tagfüggvény, amelynek feladata az objektum inicializálása (kezdeti állapotba hozása) a példányosítás pillanatában.

- **Szintaxis:** A neve megegyezik az osztály nevével, és **nincs visszatérési típusa** (még `void` sem).
- **Hívás:** Automatikusan hívódik meg, amikor az objektum létrejön (stack-en, vagy `new` operátorral a heap-en).
- **Túlterhelhető (Overloading):** Egy osztálynak több konstruktora is lehet, eltérő paraméterlistával.
- **Taginicializáló lista (Member Initializer List):**
  - A konstruktor törzse *előtt* fut le (`:` után felsorolva).
  - Ez az egyetlen mód a `const`, referencia típusú, vagy paraméter nélküli konstruktorral nem rendelkező tagok inicializálására.
  - Hatékonyabb, mint a törzsben történő értékadás (ott már a *default* konstruktor utáni felülírás történne).

18.2 A fordító által automatikusan biztosított konstruktorok

Ha a fejlesztő nem ír sajátot, a fordító (bizonyos szabályok mellett) automatikusan generálja (implicit) a következőket `public` és `inline` láthatósággal:

1. **Alapértelmezett (Default) konstruktor:**
   - **Szignatúra:** `Osztaly()` (paraméter nélküli).
   - **Működés:** Meghívja az ősosztályok és adattagok alapértelmezett konstruktorait.
   - **Szabály:** Csak akkor generálódik, ha **semmilyen** más konstruktort nem deklaráltunk.
2. **Másoló (Copy) konstruktor:**
   - **Szignatúra:** `Osztaly(const Osztaly& other)`.
   - **Működés:** Tagról-tagra másolást végez (*shallow copy*). Pointerek esetén ez veszélyes lehet (ugyanoda mutatnak).
   - **Szabály:** Generálódik, ha nem tiltjuk le, vagy nem írunk mozgató műveleteket.
3. **Mozgató (Move) konstruktor (C++11 óta):**
   - **Szignatúra:** `Osztaly(Osztaly&& other)`.
   - **Működés:** Az erőforrásokat (pl. pointereket) „átlopja” az ideiglenes objektumból a másolás helyett.
   - **Szabály:** Csak akkor generálódik, ha nincs felhasználó által deklarált másoló művelet, destruktor vagy mozgató értékadás.

18.3 Példakód

Az alábbi példa bemutatja a konstruktorok fajtáit és az inicializáló lista használatát.

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

class Data {
    int* buffer;
    const int size; // Csak listában inicializálható!

public:
    // 1. Felhasználói konstruktor (inicializáló listával)
    Data(int s) : size(s) {
        buffer = new int[size];
        std::cout << "Konstruktor: memoria foglalasa" << std::endl;
    }

    // 2. Másoló konstruktor (Deep Copy megvalósítása)
    // Ha nem írnánk meg, az alapértelmezett csak a pointert másolná!
    Data(const Data& other) : size(other.size) {
        buffer = new int[size];
        for(int i=0; i < size; i++) buffer[i] = other.buffer[i];
        std::cout << "Masolo konstruktor" << std::endl;
    }

    // 3. Mozgató konstruktor (C++11)
    Data(Data&& other) noexcept : size(other.size), buffer(other.buffer) {
        other.buffer = nullptr; // Az eredetit lenullázzuk
        std::cout << "Mozgato konstruktor" << std::endl;
    }

    ~Data() { delete[] buffer; }
};

int main() {
    Data d1(10);          // Sima konstruktor
    Data d2 = d1;         // Másoló konstruktor
    Data d3 = std::move(d1); // Mozgató konstruktor

    // Data d4; // HIBA: Nincs default ctor, mert írtunk mást!
    return 0;
}
```

19 Ismertesse a konstruktor megadásának szabályait! Milyen esetekben kell felülírnuk a fordító által definiált konstruktorokat?

19.1 A konstruktor megadásának szabályai

A konstruktorok definiálásakor a szintaktikai szabályokon túl szemantikai elveket is követnünk kell a helyes működés érdekében.

- **Név és Típus:**
  - A neve megegyezik az osztály nevével.
  - **Nincs visszatérési típusa**, még `void` sem.
- **Paraméterek és Túlterhelés:**
  - Bármennyi paramétere lehet, és túlterhelhető (overloading).
  - Használható alapértelmezett paraméterérték (default argument). Ha minden paraméternek van default értéke, az *default konstruktornak* számít.
- **Taginicializáló lista (Initializer List):**
  - A konstruktor törzse előtt, kettősponttal (`:`) bevezetve.
  - **Kötelező:** konstans adattagok, referencia típusú tagok, és paraméter nélküli konstruktorral nem rendelkező objektum-adattagok esetén.
  - **Sorrend:** A tagok inicializálása a *deklaráció sorrendjében* történik, nem a listában írt sorrendben!
- **Explicit kulcsszó:**
  - Egyparaméteres konstruktorok elé ajánlott kiírni az `explicit` kulcsszót.
  - Ez megakadályozza, hogy a fordító implicit típuskonverziót hajtson végre (pl. `int` -> `Osztaly` átalakítás egy értékadásnál).

19.2 Mikor kell felülírni a fordító által generált konstruktorokat?

Az alapértelmezett működés (sekély másolás / shallow copy) gyakran nem megfelelő. A döntést általában a **„A Három Szabálya" (Rule of Three)** – modern C++-ban az Öt Szabálya – alapján hozzuk meg.

1. **Erőforrás-kezelés (Nyers pointerek):** Ha az osztály dinamikusan foglal memóriát (`new`) a konstruktorban, és felszabadítja a destruktorban, akkor kötelező megírni a **másoló konstruktort**.
   - *Ok:* A generált másoló konstruktor csak a pointer címét másolná. Destruktor híváskor mindkét objektum ugyanazt a memóriát próbálná felszabadítani (*double free*), illetve az egyik módosítása hatna a másikra.
   - *Megoldás:* Mély másolás (Deep Copy) implementálása.
2. **Nem memória jellegű erőforrások (Fájlkezelők, Socketek):** Ha az osztály exkluzív hozzáférést birtokol (pl. megnyitott fájl), a másolás logikailag hibás lehet. Ilyenkor a másoló konstruktort gyakran **töröljük** (`= delete`), hogy megakadályozzuk a duplikálást.
3. **Validáció:** Ha az objektum csak bizonyos feltételek mellett jöhet létre (pl. egy kör sugara nem lehet negatív), a saját konstruktorban ellenőrzést (exception dobást) kell végrehajtani.
4. **Explicit inicializálás:** Ha azt szeretnénk, hogy az osztály adattagjai (pl. primitív típusok, pointerek) biztosan nullázva legyenek, és ne szemetet tartalmazzanak, mivel a default konstruktor a primitív típusokat nem inicializálja automatikusan.

19.3 Példakód

Az alábbi példa bemutatja az `explicit` használatát és a mély másolás szükségességét (Rule of Three).

```cpp
class IntBuffer {
    int* data;
    size_t size;

public:
    // 1. EXPLICIT: Megelőzi a véletlen konverziót (pl. IntBuffer b = 5;)
    explicit IntBuffer(size_t s) : size(s) {
        // Erőforrás foglalás -> Kell saját destruktor és copy ctor!
        data = new int[size];
        for(size_t i=0; i<size; ++i) data[i] = 0;
    }

    ~IntBuffer() {
        delete[] data;
    }

    // 2. MÁSOLÓ KONSTRUKTOR FELÜLÍRÁSA (Deep Copy)
    // Ha ezt nem írnánk meg, a default ctor csak a pointert másolná.
    // Eredmény: két objektum mutatna ugyanoda -> crash a delete-nél.
    IntBuffer(const IntBuffer& other) : size(other.size) {
        data = new int[size]; // Új memória foglalása
        for(size_t i=0; i<size; ++i) {
            data[i] = other.data[i]; // Adatok átmásolása
        }
    }

    // A teljesség kedvéért az értékadó operátort is illene felülírni
    // IntBuffer& operator=(const IntBuffer& other) { ... }
};

void processBuffer(const IntBuffer& buf) { /* ... */ }

int main() {
    IntBuffer b1(10);

    // processBuffer(10); // HIBA: Az 'explicit' miatt nem konvertál int-et
    processBuffer(IntBuffer(10)); // OK: Explicit hívás

    IntBuffer b2 = b1; // Saját másoló konstruktor hívódik (biztonságos)
    return 0;
}
```

20 Ismertesse az adattagok kezdeti értékmegadásának lehetőségeit! Ezek közül melyik az, amelyik referencia típusú adattagok esetén használható?

20.1 Az értékmegadás lehetőségei

C++-ban három fő módja van az osztály adattagjainak kezdeti értékkel való ellátására. Fontos különbséget tenni az *inicializáció* és az *értékadás* között.

1. **Taginicializáló lista (Member Initializer List):**
   - A konstruktor paraméterlistája után, kettősponttal (`:`) elválasztva.
   - Ez a valódi inicializáció helye: az értékadás még az objektum memóriaterületének létrejöttekor történik, a konstruktor törzse *előtt*.
   - A leggyorsabb és leghatékonyabb módszer.
2. **Értékadás a konstruktor törzsében:**
   - A változó először létrejön (lefut a default konstruktora), majd a törzsben kap új értéket.
   - Technikailag ez nem inicializálás, hanem felülírás.
   - Lassabb lehet összetett objektumoknál (felesleges default konstruktor hívás + értékadás operátor).
3. **Osztályon belüli inicializálás (In-class member initialization - C++11):**
   - A deklarációval egy helyen adunk alapértéket (pl. `int x = 0;`).
   - Ha a konstruktor inicializáló listájában is szerepel a változó, az felülírja ezt az alapértelmezett értéket.

20.2 Referencia típusú adattagok kezelése

A referencia típusú (`&`) adattagok esetén **kizárólag a Taginicializáló lista** (vagy ritkábban az osztályon belüli inicializálás) használható.

- **Ok:** A referenciát a létrehozása pillanatában „hozzá kell kötni” valamihez.
- Nem létezhet „üres” referencia, amit később állítunk be.
- A konstruktor törzsében történő értékadás már túl késő lenne, mert addigra a tagnak léteznie kellene (de inicializáló lista nélkül nem tud létrejönni).
- Ugyanez a szabály vonatkozik a `const` adattagokra is.

20.3 Példakód

Az alábbi példa bemutatja a helyes (inicializáló lista) és a helytelen (törzsben értékadás) használatot referencia esetén.

```
class Wrapper {
private:
    int& refMember; // Referencia adattag
    int valMember;  // Sima adattag

public:
    // HELYES MEGOLDÁS:
    // A 'refMember' kötése a listában történik.
    Wrapper(int& target, int value)
```

```
10          : refMember(target), // Kötelező itt!
11            valMember(value)   // Opcionális, de ajánlott itt
12      {
13          // A konstruktor törzse
14      }
15
16      /* HIBÁS MEGOLDÁS (Compile Error):
17
18      Wrapper(int& target, int value) {
19          refMember = target; // HIBA!
20          // Itt a refMember-nek már léteznie kellene,
21          // de nincs mihez kötve. Ráadásul referenciát
22          // nem lehet átirányítani (rebind).
23
24          valMember = value; // Ez működne, de nem hatékony.
25      }
26      */
27  };
28
29  int main() {
30      int x = 10;
31      Wrapper w(x, 5); // A w.refMember mostantól x-re hivatkozik
32      return 0;
33  }
```

21 Ismertesse példával az 1 paraméterrel rendelkező konstruktor egyszerűsített meghívási lehetőségét! Hogyan tudjuk ezt az egyszerűsítést letiltani?

21.1 Egyszerűsített meghívás (Implicit konverzió)

Ha egy osztály konstruktora egyetlen paraméterrel hívható meg (vagy egy paraméteres, vagy a többi paraméternek van alapértelmezett értéke), a C++ fordító ezt **konverziós konstruktornak** tekinti.

- **Másoló inicializálás (Copy Initialization):** Lehetővé teszi az egyenlőségjel (`=`) használatát objektum létrehozásakor, mintha csak egy primitív típust adnánk értékül.
- **Paraméter átadás:** Ha egy függvény paramétere az adott osztály típusú, de mi a konstruktor paraméterének megfelelő típust (pl. `int`) adunk át, a fordító automatikusan létrehoz egy ideiglenes objektumot.
- **Veszélye:** Ez a működés néha nem kívánt, véletlen típuskonverziókhoz és nehezen felderíthető hibákhoz vezethet.

21.2 Az egyszerűsítés letiltása (explicit)

A fenti automatizmust az `explicit` kulcsszóval tilthatjuk le.

- **Használat:** A konstruktor deklarációja elé kell írni az `explicit` szót.
- **Hatása:** Megtiltja az implicit konverziót és a `=` jellel történő inicializálást.
- **Következmény:** Az objektumot csak direkt módon (zárójellel vagy kapcsos zárójellel) lehet inicializálni.

21.3 Példakód

Az alábbi példa bemutatja a különbséget egy implicit (hagyományos) és egy `explicit` módon védett osztály között.

```cpp
#include <iostream>

// 1. Implicit konverziót engedő osztály
class Celsius {
public:
    double value;
    // Nincs 'explicit', a fordító automatikusan konvertálhat double-ből
    Celsius(double v) : value(v) {}
};

// 2. Implicit konverziót tiltó osztály
class Fahrenheit {
public:
    double value;
    // 'explicit': csak direkt hívás engedélyezett!
    explicit Fahrenheit(double v) : value(v) {}
};

void printCelsius(Celsius c) {
    std::cout << c.value << " C" << std::endl;
}

```

```cpp
void printFahrenheit(Fahrenheit f) {
    std::cout << f.value << " F" << std::endl;
}

int main() {
    // --- Implicit eset (Celsius) ---

    // Működik: egyszerűsített (másoló) inicializálás
    Celsius c1 = 25.0;

    // Működik: implicit konverzió függvényhíváskor
    // A fordító látja, hogy 10.0 (double) -> Celsius konstruktor létezik
    printCelsius(10.0);

    // --- Explicit eset (Fahrenheit) ---

    // HIBA: "conversion from double to non-scalar type requested"
    // Fahrenheit f1 = 80.0;

    // HIBA: a függvény Fahrenheit-et vár, de double-t kapott,
    // és az automatikus átalakítás tiltva van.
    // printFahrenheit(80.0);

    // HELYES HASZNÁLAT explicit konstruktornál:
    Fahrenheit f2(80.0);       // Direkt inicializálás
    Fahrenheit f3 = Fahrenheit(80.0); // Explicit konverzió
    printFahrenheit(Fahrenheit(80.0)); // Ideiglenes obj. létrehozása

    return 0;
}
```

22 Ismertesse a másoló konstruktor megírásának szükségességét okozó szituációt! Honnan tudjuk eldönteni, hogy a fordító a másoló konstruktort, vagy az „=” operátort használja?

22.1 Mikor szükséges saját másoló konstruktort írni?

Alapértelmezésben a fordító egy úgynevezett *sekély másolatot* (shallow copy) készítő konstruktort generál, amely bitről-bitre lemásolja az adattagokat. Ez bizonyos esetekben végzetes hibát okoz.

- **A problémás szituáció (Nyers pointerek):** Ha az osztály egy pointert tartalmaz, amely dinamikusan foglalt memóriára (heap) mutat, a sekély másolás csak a pointer címét másolja át, nem a mögötte lévő adatot.
- **Következmények:**
  1. **Osztozott erőforrás:** Két különböző objektum ugyanazt a memóriaterületet használja és módosítja véletlenül.
  2. **Double Free hiba:** Amikor az objektumok megszűnnek (scope vége), mindkettő megpróbálja felszabadítani (`delete`) ugyanazt a memóriacímet. Az első sikerül, a második programösszeomlást (crash) okoz.
- **Megoldás (Mély másolás / Deep Copy):** Saját másoló konstruktort kell írni, amely:
  1. Új memóriaterületet foglal az új objektumnak.
  2. Átmásolja az *értékeket* az eredeti területről az újra.

  Megjegyzés: Ez a „Rule of Three” (Három Szabálya) egyik alappillére.

22.2 Másoló konstruktor vs. Értékadó operátor

Bár mindkét művelet során adatok kerülnek egyik objektumból a másikba, és szintaktikailag mindkettőben szerepelhet az egyenlőségjel, a különbség az objektum **létrejöttének állapotában** keresendő.

- **Másoló konstruktor (Copy Constructor):**
  - **Mikor:** Amikor egy **új** objektumot hozunk létre egy már létező mintájára.
  - **Kulcs:** Az objektum a sor végrehajtása előtt még nem létezett.
  - **Szintaxis:** `Osztaly A = B;` vagy `Osztaly A(B);`
- **Értékadó operátor (Assignment Operator):**
  - **Mikor:** Amikor egy **már létező, inicializált** objektum értékét írjuk felül egy másikéval.
  - **Kulcs:** Az objektum már létezik a memóriában, és van érvényes (vagy érvénytelen) állapota, amit előbb esetleg takarítani kell.
  - **Szintaxis:** `A = B;` (ahol A-t korábban már deklaráltuk).

22.3 Példakód

Az alábbi példa bemutatja a mély másolás szükségességét és a hívások megkülönböztetését.

```
1 #include <iostream>
2
3 class Buffer {
4     int* data;
5
```

```
public:
    // Sima konstruktor
    Buffer(int value) {
        data = new int(value);
        std::cout << "Konstruktor (new)" << std::endl;
    }

    // SAJÁT MÁSOLÓ KONSTRUKTOR (Deep Copy)
    // Szükséges, különben a pointer címe másolódna!
    Buffer(const Buffer& other) {
        data = new int(*other.data); // Új memória, érték másolása
        std::cout << "Masolo Konstruktor (Deep Copy)" << std::endl;
    }

    // ÉRTÉKADÓ OPERÁTOR
    Buffer& operator=(const Buffer& other) {
        if (this != &other) { // Ön-értékadás ellenőrzése
            delete data;      // Régi adat törlése (FONTOS különbség!)
            data = new int(*other.data); // Új adat másolása
        }
        std::cout << "Ertekado operator (=)" << std::endl;
        return *this;
    }

    ~Buffer() { delete data; }
};

int main() {
    Buffer b1(10); // Sima konstruktor

    // 1. ESET: MÁSOLÓ KONSTRUKTOR
    // Új objektum (b2) jön létre b1 mintájára.
    // A fordító ezt látja: Buffer b2(b1);
    Buffer b2 = b1;

    Buffer b3(20); // Sima konstruktor

    // 2. ESET: ÉRTÉKADÓ OPERÁTOR
    // A b3 objektum már létezik! Csak az értékét cseréljük le.
    b3 = b1;

    return 0;
}
```

23 Ismertesse a destruktor definícióját, a destruktor készítés szabályait! Mit mondhatunk a destruktor kézi meghívásáról?

23.1 Definíció és Szerep

A destruktor egy speciális tagfüggvény, amely akkor hívódik meg automatikusan, amikor egy objektum élettartama véget ér (megszűnik).

- **Célja:** Az objektum által lefoglalt erőforrások felszabadítása (takarítás).
- **Tipikus feladatok:** Dinamikus memória felszabadítása (`delete`), nyitott fájlok bezárása, hálózati kapcsolatok bontása, mutexek elengedése.
- **Jele:** A neve megegyezik az osztály nevével, de egy hullámvonal (`~`) előzi meg.

23.2 A készítés szabályai

A destruktorra szigorú szintaktikai és szemantikai szabályok vonatkoznak:

- **Paraméterek:** Nem rendelkezhet paraméterrel. Ebből következik, hogy a destruktor **nem terhelhető túl** (overloading), osztályonként csak egy lehet belőle.
- **Visszatérési érték:** Nincs visszatérési típusa, még `void` sem.
- **Kivételbiztonság:** Szigorúan tilos kivételt (`throw`) kiengedni a destruktorból.
  - *Ok:* Ha a destruktor egy másik kivétel miatti „stack unwinding” (veremlebontás) során fut le, és dob még egy kivételt, a program azonnal összeomlik (`std::terminate`).
- **Virtuális destruktor:**
  - Ha egy osztályt ősosztálynak szánunk (polimorfizmus), a destruktornak `virtual`-nak kell lennie.
  - *Ok:* Ha az ősosztály pointerén keresztül törlünk egy leszármazott objektumot, és nem virtuális a destruktor, csak az ős része semmisül meg (Undefined Behavior / Memory Leak).

23.3 A destruktor kézi meghívásáról

Normál körülmények között a destruktort **soha nem hívjuk meg** explicit módon.

- **Általános szabály:** A fordító automatikusan beszúrja a hívást a blokk végére (stack objektumok) vagy a `delete` operátor hívásakor (heap objektumok).
- **Veszélye:** Ha kézzel meghívjuk, majd az objektum megszűnésekor automatikusan újra lefut, az *Double Free* hibához és programösszeomláshoz vezet.
- **Az egyetlen kivétel (Placement New):**
  - Ha az objektumot a *Placement New* segítségével egy előre lefoglalt memóriaterületre hoztuk létre (anélkül, hogy a memóriát az operációs rendszertől kértük volna), akkor nem hívhatunk rá `delete`-et (mert nem szabad felszabadítani a puffert).
  - Ilyenkor **kötelező** kézzel meghívni a destruktort a takarításhoz.

23.4 Példakód

Az alábbi példa bemutatja a helyes destruktor definíciót és a ritka kivételt, a kézi meghívást.

```cpp
#include <iostream>
#include <new> // placement new-hez

class Resource {
    int* data;
public:
    Resource() {
        data = new int[100];
        std::cout << "Konstruktor: memoria foglalva" << std::endl;
    }

    // Virtuális destruktor (ha esetleg örökölnénk belőle)
    virtual ~Resource() {
        delete[] data;
        std::cout << "Destruktor: memoria felszabaditva" << std::endl;
    }
};

int main() {
    // --- 1. Normál eset (Automatikus hívás) ---
    {
        Resource r;
    } // Itt automatikusan lefut a ~Resource()

    // --- 2. Speciális eset (Placement New és Kézi hívás) ---

    // Nyers memória puffer (stack-en)
    alignas(Resource) char buffer[sizeof(Resource)];

    // Objektum létrehozása a pufferben (nem allokál új memóriát)
    Resource* ptr = new(buffer) Resource();

    // TILOS: delete ptr;
    // Mivel a 'buffer' a stack-en van, nem szabad free-t hívni rá!

    // HELYES: Kézi destruktor hívás
    ptr->~Resource();

    return 0;
}
```

24 Ismertesse a névterek definiálásának szükségességét a C++ programokban! Melyik operátorral hivatkozhatunk egy adott névtérben található osztályra?

24.1 A névterek (Namespaces) szükségessége

A C++ programozásban a névterek elsődleges célja a globális névtérben fellépő zsúfoltság és a **névütközések (name collisions)** megakadályozása.

- **Névütközések elkerülése:** Nagyobb szoftverrendszerek vagy több külső könyvtár (library) használata esetén gyakori, hogy azonos neveket használnak (pl. `Node`, `String`, `Vector`). Névterek nélkül a fordító nem tudná megkülönböztetni ezeket, ami fordítási hibát okozna.
- **Logikai szervezés:** A névterek lehetővé teszik a kód moduláris felépítését. A kapcsolódó osztályokat és függvényeket (pl. fájlkezelés, hálózat, grafika) logikailag elkülönített csoportokba rendezhetjük (pl. `std`, `boost`, `sfml`).
- **Globális névtér védelme:** Megakadályozza a globális változók és függvények véletlen felülírását vagy árnyékolását.

24.2 Hivatkozás az elemekre

Egy adott névtérben található osztályra vagy tagra a **hatókör-feloldó operátorral (Scope Resolution Operator)** hivatkozhatunk.

- **Jele:** `::` (kettős kettőspont).
- **Formátum:** `NévtérNeve::Azonosító`.
- **Alternatíva (using):** A `using namespace ...;` utasítással a névtér elemei beemelhetők az aktuális hatókörbe, így az operátor elhagyható, de ez a névütközések veszélye miatt óvatosan használandó.

24.3 Példakód

Az alábbi példa két azonos nevű osztály (`Connection`) békés együttélését mutatja be névterek segítségével.

```cpp
#include <iostream>

// 1. Hálózati modul névtere
namespace Network {
    class Connection {
    public:
        void connect() { std::cout << "Connecting via TCP..." << std::endl; }
    };
}

// 2. Adatbázis modul névtere
namespace Database {
    class Connection {
    public:
        void connect() { std::cout << "Connecting to SQL..." << std::endl; }
    };
}

int main() {
    // Névtér minősítés nélkül fordítási hiba lenne:
    // Connection c; // HIBA: "ambiguous"
```

```
22
23      // Használat a :: operátorral
24      Network::Connection netConn;
25      Database::Connection dbConn;
26
27      netConn.connect();
28      dbConn.connect();
29
30      // Using deklaráció egy adott elemre
31      using Network::Connection;
32      Connection c; // Most a Network::Connection-t jelenti
33
34      return 0;
35  }
```

25 Ismertesse az osztálypéldányokon végzett műveletek definiálási lehetőségeit! Mely műveleteket nem lehet átdefiniálni?

25.1 Az operátor-túlterhelés (Operator Overloading) lehetőségei

C++-ban az operátorok (pl. +, -, ==) új jelentést kaphatnak felhasználói típusok (osztályok) esetén. Két alapvető módon definiálhatjuk őket:

1. **Tagfüggvényként (Member Function):**
   - Az operátor az osztály része.
   - **Bal oldali operandus:** Implicit módon mindig az aktuális objektum (`*this`).
   - **Paraméterek száma:** Eggyel kevesebb, mint az operandusok száma (bináris operátornál 1 paraméter, unárisnál 0).
   - **Kötelező így írni:** `=`, `[]`, `()`, `->`.
2. **Globális (szabad) függvényként (Global Function):**
   - Az osztályon kívül definiáljuk.
   - **Bal oldali operandus:** Az első paraméterként adjuk át (nincs `this`).
   - **Barát (friend) státusz:** Gyakran szükséges, hogy a függvény hozzáférjen a privát adattagokhoz.
   - **Előnye:** Lehetővé teszi a szimmetrikus konverziót (pl. `10 + obj` és `obj + 10` is működhet).

25.2 Nem átdefiniálható operátorok

A nyelv védelme érdekében bizonyos operátorok működése rögzített, ezeket **tilos** túlterhelni:

- `.` (Pont operátor / Tagkiválasztás)
- `::` (Hatókör-feloldó / Scope resolution)
- `?:` (Feltételes / Ternary operátor)
- `sizeof` (Méret lekérdezése)
- `typeid` (Típusinformáció)
- `.*` (Tagra mutató pointer feloldása)

25.3 Példakód

Az alábbi példa bemutatja az összeadás (`+`) globális barátként, és az értékadás (`+=`) tagfüggvényként történő megvalósítását.

```cpp
#include <iostream>

class Vector2 {
private:
    int x, y;

public:
    Vector2(int x, int y) : x(x), y(y) {}

    // 1. Tagfüggvényként definiált operátor (+=)
    // A bal oldali operandus a 'this', a jobb oldali az 'other'
    // Módosítja az objektum állapotát.
```

```
13      Vector2& operator+=(const Vector2& other) {
14          this->x += other.x;
15          this->y += other.y;
16          return *this; // Láncolhatóság miatt referenciával térünk vissza
17      }
18
19      // 2. Globális (Barát) függvényként definiált operátor (+)
20      // Két paramétert kap, új objektumot hoz létre.
21      friend Vector2 operator+(const Vector2& lhs, const Vector2& rhs);
22
23      void print() const { std::cout << x << "," << y << std::endl; }
24  };
25
26  // Globális definíció
27  Vector2 operator+(const Vector2& lhs, const Vector2& rhs) {
28      // Nem módosítjuk a paramétereket, új példányt adunk vissza
29      return Vector2(lhs.x + rhs.x, lhs.y + rhs.y);
30  }
31
32  int main() {
33      Vector2 v1(1, 2);
34      Vector2 v2(3, 4);
35
36      Vector2 v3 = v1 + v2; // Globális operator+ hívása
37      v1 += v2;             // Tagfüggvény operator+= hívása
38
39      v3.print(); // 4,6
40      v1.print(); // 4,6
41      return 0;
42  }
```

26 Ismertesse az osztályok kétoperandusú műveleteinek átdefiniálási lehetőségeit! Írjon példákat minden egyes lehetőséghez!

26.1 Áttekintés

Kétoperandusú (bináris) operátorok (pl. `+`, `-`, `*`, `==`) esetén az operátornak egy bal oldali (LHS) és egy jobb oldali (RHS) operandusa van. C++-ban kétféleképpen definiálhatjuk ezeket.

26.2 1. Lehetőség: Tagfüggvényként (Member Function)

Az operátort az osztályon belül definiáljuk.

- **Paraméterek száma:** Csak egy (a jobb oldali operandus).
- **Bal oldali operandus:** Implicit módon az aktuális objektum (`*this`).
- **Korlát:** A bal oldali operandusnak mindenképpen az adott osztály típusának kell lennie. Ezért például a `10 + obj` kifejezés nem valósítható meg így (mert a `int`-nek nincs ilyen tagfüggvénye).
- **Const-correctness:** Ha az operátor nem módosítja az objektumot (pl. összeadás), a függvényt `const`-ként kell jelölni.

26.3 2. Lehetőség: Globális (Barát) függvényként (Global/Friend Function)

Az operátort az osztályon kívül, szabad függvényként definiáljuk.

- **Paraméterek száma:** Kettő (bal oldali és jobb oldali operandus).
- **Hozzáférési jog:** Ha a függvénynek el kell érnie a privát adattagokat, az osztályon belül `friend` kulcsszóval kell deklarálni.
- **Előny (Szimmetria):** Lehetővé teszi az implicit típuskonverziót a bal oldali operanduson is. (Pl. ha van `int` -> `Osztaly` konverzió, akkor a `10 + obj` működni fog).
- **Kötelező használat:** Ha a bal oldali operandus nem a mi osztályunk (pl. `std::ostream` a `«` operátornál).

26.4 Példakód

Az alábbi példa a kivonást (`-`) tagfüggvényként, az összeadást (`+`) és a kiíratást (`«`) pedig globális barát függvényként valósítja meg.

```
#include <iostream>

class Pont {
private:
    int x, y;

public:
    Pont(int x = 0, int y = 0) : x(x), y(y) {}

    // --- 1. LEHETŐSÉG: Tagfüggvény (Kivonás) ---
    // Hívás: p1 - p2
    // Bal oldal: this, Jobb oldal: other
    Pont operator-(const Pont& other) const {
        // Új objektumot adunk vissza, az eredetit nem módosítjuk
        return Pont(this->x - other.x, this->y - other.y);
    }
```

```cpp

    // --- 2. LEHETŐSÉG: Globális Barát Függvények ---

    // Összeadás (Barát deklaráció)
    // Hívás: p1 + p2 vagy p1 + 10 (konverzióval)
    friend Pont operator+(const Pont& lhs, const Pont& rhs);

    // Kiíratás (Barát deklaráció)
    // Kötelező globálisnak lennie, mert a bal oldal std::ostream
    friend std::ostream& operator<<(std::ostream& os, const Pont& p);
};

// Globális függvények definíciója:

Pont operator+(const Pont& lhs, const Pont& rhs) {
    // Itt nincs 'this', mindkét paraméter explicit
    return Pont(lhs.x + rhs.x, lhs.y + rhs.y);
}

std::ostream& operator<<(std::ostream& os, const Pont& p) {
    os << "(" << p.x << ", " << p.y << ")";
    return os;
}

int main() {
    Pont p1(10, 20);
    Pont p2(5, 5);

    // Tagfüggvény hívása
    Pont p3 = p1 - p2;

    // Globális függvény hívása
    Pont p4 = p1 + p2;

    // Globális operátor, láncolva
    std::cout << p3 << " es " << p4 << std::endl;

    return 0;
}
```

27 Ismertesse az osztályok egyoperandusú műveleteinek átdefiniálási lehetőségeit! Írjon példákat minden egyes lehetőséghez!

27.1 Áttekintés

Az egyoperandusú (unáris) operátorok (pl. `++`, `-`, `-` (negálás), `!`) egyetlen objektumon fejtik ki hatásukat. Ezen operátorok túlterhelésére két fő lehetőség van, illetve egy speciális szabály vonatkozik a prefix/postfix megkülönböztetésre.

27.2 1. Lehetőség: Tagfüggvényként (Member Function)

Ha az operátort az osztály tagjaként definiáljuk:

- **Paraméterek száma:** Általában **0 paramétere** van.
- **Operandus:** Az operátor implicit módon az aktuális objektumon (`*this`) hajtódik végre.
- **Előnye:** Közvetlenül hozzáfér a privát adattagokhoz, nem kell `friend` deklaráció.

27.3 2. Lehetőség: Globális (Barát) függvényként (Global Function)

Ha az operátort az osztályon kívül definiáljuk:

- **Paraméterek száma: 1 paramétere** van (az osztály típusú objektum referenciája).
- **Operandus:** A függvény paraméterként kapja meg az objektumot.
- **Használat:** Gyakran `friend`-ként deklarálják az osztályban a hozzáférés miatt.

27.4 Speciális eset: Prefix vs. Postfix (++ és −)

Mivel a `++a` (prefix) és `a++` (postfix) operátorok ugyanazt a jelet használják, a C++ egy mesterséges paraméterrel különbözteti meg őket:

- **Prefix (Előtag):** Nincs paraméter (vagy globálisnál 1 db referencia). Referenciával tér vissza a módosított objektumra.
- **Postfix (Utótag):** Egy **fiktív `int` paramétert** kap. Értékkel tér vissza (a módosítás előtti állapottal).

27.5 Példakód

Az alábbi példa bemutatja a három leggyakoribb esetet: a negálást globális függvényként, valamint a prefix és postfix inkrementálást tagfüggvényként.

```
#include <iostream>

class Szam {
private:
    int ertek;

public:
    Szam(int v = 0) : ertek(v) {}

    // 1. ESET: Prefix inkrementálás (++obj) TAGFÜGGVÉNYKÉNT
    // Nincs paraméter.
    // Előbb növelünk, aztán visszaadjuk önmagát referenciaként.
    Szam& operator++() {
```

```
14          this->ertek += 1;
15          return *this;
16      }
17
18      // 2. ESET: Postfix inkrementálás (obj++) TAGFÜGGVÉNYKÉNT
19      // A "dummy" int paraméter jelzi a fordítónak, hogy ez postfix.
20      // Elmentjük a régit, növelünk, visszaadjuk a régit érték szerint.
21      Szam operator++(int) {
22          Szam regi = *this; // Másolat készítése
23          this->ertek += 1;  // Növelés
24          return regi;       // Régi érték visszaadása
25      }
26
27      // Kiíratáshoz
28      void print() const { std::cout << "Ertek: " << ertek << std::endl; }
29
30      // Barát deklaráció a globális negáláshoz
31      friend Szam operator-(const Szam& sz);
32  };
33
34  // 3. ESET: Negálás (-obj) GLOBÁLIS FÜGGVÉNYKÉNT
35  // Egy paramétert kap. Új objektumot ad vissza.
36  Szam operator-(const Szam& sz) {
37      return Szam(-sz.ertek);
38  }
39
40  int main() {
41      Szam n(10);
42
43      ++n;       // Prefix hívás (most 11)
44      n.print();
45
46      n++;       // Postfix hívás (most 12, de a kifejezés értéke 11 volt)
47      n.print();
48
49      Szam neg = -n; // Globális unáris mínusz hívása (-12)
50      neg.print();
51
52      return 0;
53  }
```

28 Ismertesse a kommutatív műveletek átdefiniálási lehetőségét! Miért nem tudjuk a tagfüggvényes módszert alkalmazni?

28.1 A probléma: Miért nem jó a tagfüggvény?

A kommutativitás (felcserélhetőség) azt jelenti, hogy a művelet eredménye független az operandusok sorrendjétől (pl. $A + B = B + A$). Vegyes típusú műveleteknél (pl. `Objektum` és `int`) a tagfüggvényes megközelítés aszimmetrikus.

- **Bal oldali kötöttség:** Tagfüggvényként definiált operátor esetén a bal oldali operandusnak (`LHS`) **kötelezően** az osztály típusának kell lennie.
- **A hívás mechanizmusa:** A fordító a `obj + 10` kifejezést `obj.operator+(10)` formára fordítja.
- **A hiba:** Ha megfordítjuk a sorrendet (`10 + obj`), a fordító a `10.operator+(obj)` hívást keresné. Mivel az `int` egy beépített primitív típus, nem rendelkezik tagfüggvényekkel, így ez fordítási hibát okoz.
- **Következtetés:** Tagfüggvényként csak akkor működne a művelet, ha az osztályunk a bal oldalon áll.

28.2 A megoldás: Globális (Barát) függvény

A teljes kommutativitás eléréséhez az operátort az osztályon kívül, globális függvényként kell definiálni.

- **Szimmetria:** Globális függvény esetén mindkét operandus egyenrangú paraméterként jelenik meg: `operator+(LHS, RHS)`.
- **Implicit konverzió:** Ha az operátor mindkét paramétere az osztály típusát várja, és az osztály rendelkezik megfelelő (nem `explicit`) konstruktorral, a fordító képes a primitív típust (pl. `int`) automatikusan átalakítani objektummá.
- **Eredmény:** Így mind a `obj + 10`, mind a `10 + obj` működni fog.

28.3 Példakód

Az alábbi példa egy `Number` osztályt mutat be, amely összeadható `int`-tel mindkét irányból.

```cpp
#include <iostream>

class Number {
    int value;
public:
    // Konstruktor (implicit konverziót engedélyez int-ből)
    Number(int v) : value(v) {}

    int getValue() const { return value; }

    // HIBÁS MEGKÖZELÍTÉS (Tagfüggvény):
    // Number operator+(const Number& other) const { ... }
    // Ez csak a (Number + int) esetet fedné le, az (int + Number)-t NEM.

    // HELYES MEGKÖZELÍTÉS (Globális Barát):
    friend Number operator+(const Number& lhs, const Number& rhs);
};

// Globális operátor definíció
// Mivel mindkét paraméter Number típusú, a fordító
```

```cpp
// implicit konverziót végez, ha int-et lát bármelyik oldalon.
Number operator+(const Number& lhs, const Number& rhs) {
    return Number(lhs.value + rhs.value);
}

int main() {
    Number n(5);

    // 1. eset: Objektum + int
    // A fordító átalakítja: operator+(n, Number(10))
    Number res1 = n + 10;

    // 2. eset: int + Objektum (Kommutativitás)
    // A fordító átalakítja: operator+(Number(10), n)
    // Ez tagfüggvénnyel lehetetlen lenne!
    Number res2 = 10 + n;

    std::cout << res1.getValue() << " " << res2.getValue() << std::endl;
    return 0;
}
```

29 Ismertesse a „()” operátor túlterhelési lehetőségeit!

29.1 Áttekintés és Tulajdonságok

A függvényhívás operátor `operator()` túlterhelésével hozhatjuk létre az úgynevezett **funktorokat** (function objects). Ez lehetővé teszi, hogy egy objektumpéldányt úgy használjunk, mintha az egy függvény lenne.

- **Kizárólag tagfüggvényként:** A `()` operátor csak nem-statikus tagfüggvényként definiálható, globális függvényként nem.
- **Tetszőleges paraméterszám:** Ez az egyetlen operátor a C++-ban, amely tetszőleges számú $(0, 1, 2, \ldots, n)$ paramétert fogadhat. Emiatt gyakran használják többdimenziós tömbök (pl. mátrixok) indexelésére, mivel a `[]` operátor (hagyományosan) csak egy paramétert fogadhat.
- **Állapotmegőrzés (Stateful):** A hagyományos függvényekkel ellentétben a funktorok rendelkezhetnek belső állapottal (adattagokkal), amelyek megőrződnek a hívások között.
- **Túlterhelhetőség:** Egy osztályon belül többször is definiálható eltérő paraméterlistával (overloading).

29.2 Gyakori felhasználási területek

1. **Paraméterezhető műveletek:** Olyan „függvények” létrehozása, amelyek viselkedése konstruktorban állítható be (pl. egy számláló vagy egy küszöbérték-vizsgáló).
2. **STL algoritmusok:** A `std::sort`, `std::for_each` és hasonló algoritmusok gyakran várnak funktorokat predikátumként.
3. **Mátrix-kezelés:** `matrix(sor, oszlop)` formátumú elérés biztosítása.

29.3 Példakód

Az alábbi példa egy lineáris transzformációt ($y = ax + b$) megvalósító funktort mutat be, ahol az a és b paraméterek az objektum állapotát képezik.

```cpp
#include <iostream>

class LinearTransform {
private:
    // Belső állapot (State)
    double slope;     // a
    double intercept; // b

public:
    // Konstruktor: beállítja a működési paramétereket
    LinearTransform(double a, double b) : slope(a), intercept(b) {}

    // Az operátor túlterhelése
    // Tetszőleges visszatérési érték és paraméterezés lehetséges
    double operator()(double x) const {
        return (slope * x) + intercept;
    }
};

class Matrix {
    int data[10][10];
public:
    // Példa több paraméteres használatra (Mátrix indexelés)
    // A [] operátorral ezt nem lehetne így (több paraméterrel) megoldani.
```

```
25      int& operator()(int row, int col) {
26          return data[row][col];
27      }
28  };
29
30  int main() {
31      // 1. Funktor példányosítása (a=2, b=3)
32      LinearTransform func(2.0, 3.0);
33
34      // 2. Használat függvényhívás szintaxissal
35      // A fordító ezt hívja: func.operator()(5.0)
36      double result = func(5.0); // 2 * 5 + 3 = 13
37
38      std::cout << "Eredmény: " << result << std::endl;
39
40      // 3. Mátrix példa
41      Matrix m;
42      m(1, 2) = 42; // Írás a (1,2) pozícióra
43
44      return 0;
45  }
```

30 Ismertesse az „=” operátor túlterhelésének szintaktikáját és a szituációt, amelyben a fordító által biztosított operátor nem működik megfelelően!

30.1 Szintaktikai szabályok

Az értékadó operátor (`operator=`) túlterhelésére szigorú szabályok vonatkoznak, mivel alapvető nyelvi elemet módosítunk.

- **Csak tagfüggvény lehet:** Az értékadó operátort **tilos** globális (barát) függvényként definiálni. Mindenképpen az osztály nem statikus tagfüggvényének kell lennie.
- **Szignatúra:** A konvenció szerint a következő formát követi:
  - **Visszatérési érték:** Referencia az osztály típusára (`Osztaly&`). Erre a **láncolhatóság** miatt van szükség (pl. `a = b = c;`).
  - **Paraméter:** Konstans referencia a forrás objektumra (`const Osztaly& other`).
- **Ön-értékadás figyelése:** A függvény törzsének elején ellenőrizni kell, hogy az objektum saját magát kapta-e értékül (`this != &other`). Enélkül erőforrás-kezelési hiba léphet fel (töröljük az adatot, mielőtt lemásolnánk).

30.2 A fordító által biztosított operátor problémája

Ha nem írunk sajátot, a fordító generál egy alapértelmezett értékadó operátort.

- **Működése:** *Sekély másolást* (shallow copy) végez, azaz az adattagok értékeit bitről-bitre átmásolja (tagról tagra értékadás).
- **A kritikus szituáció (Pointerek):** Ha az osztály dinamikusan foglalt memóriát kezel (nyers pointer adattag), a sekély másolás csak a pointerek memóriacímét másolja át.
- **A hiba következményei:**
  1. **Memóriaszivárgás (Memory Leak):** A bal oldali objektum (amely felülíródik) korábbi pointere elveszik anélkül, hogy felszabadítottuk volna a hozzá tartozó memóriát.
  2. **Osztozott birtoklás:** Két objektum ugyanarra a memóriaterületre mutat. Ha az egyik módosítja, a másik is változik.
  3. **Double Free:** Amikor az objektumok megszűnnek, mindkét destruktor megpróbálja felszabadítani (`delete`) ugyanazt a címet, ami programösszeomlást okoz.

30.3 Megvalósítási minta (Idiómák)

A helyes implementációnak négy lépése van:

1. Ön-értékadás vizsgálata.
2. A régi (bal oldali) erőforrás felszabadítása.
3. Új memória foglalása és az adat másolása (Mély másolás).
4. Referencia visszaadása (`*this`).

30.4 Példakód

Az alábbi példa a helyes implementációt mutatja dinamikus memóriakezelés esetén.

```
class StringHolder {
private:
    char* str;

public:
    StringHolder(const char* s) {
        // ... konstruktor implementáció (memória foglalás) ...
    }

    // Értékadó operátor felülírása
    StringHolder& operator=(const StringHolder& other) {
        // 1. LÉPÉS: Ön-értékadás vizsgálata
        // Ha a két objektum címe megegyezik, nincs teendő.
        // Enélkül a 2. lépésben törölnénk azt az adatot,
        // amit a 3. lépésben másolni akarnánk!
        if (this == &other) {
            return *this;
        }

        // 2. LÉPÉS: Régi erőforrás takarítása
        // Mivel ez az objektum már létezik, lehet benne adat.
        delete[] str;

        // 3. LÉPÉS: Mély másolás (Deep Copy)
        if (other.str) {
            // Új tárterület kérése
            int len = std::strlen(other.str);
            str = new char[len + 1];
            // Adatok átmásolása
            std::strcpy(str, other.str);
        } else {
            str = nullptr;
        }

        // 4. LÉPÉS: Visszatérés önmagunkkal
        return *this;
    }

    ~StringHolder() { delete[] str; }
};
```

31 Ismertesse a „new” és „delete” operátorok túlterhelésének szabályait!

A C++ nyelvben a memóriakezelés testreszabható a `new` és `delete` operátorok túlterhelésével. Fontos különbséget tenni a *new kifejezés* (amely memóriát foglal és konstruktort hív) és az *operator new* (amely csak a nyers memóriát foglalja) között. Túlterhelni csak az utóbbit lehet.

31.1 Alapvető szintaxis és szignatúrák

Az operátorok túlterhelésekor szigorú előírások vonatkoznak a függvények szignatúrájára.

- **operator new:**
  - Visszatérési értéke kötelezően `void*`.
  - Első paramétere kötelezően `size_t` típusú (a szükséges bájtok száma).
  - További paraméterek is megadhatók (pl. *placement new* esetén), de az első a méret marad.
- **operator delete:**
  - Visszatérési értéke `void`.
  - Első paramétere kötelezően `void*` (a felszabadítandó terület mutatója).
  - Opcionálisan átveheti a `size_t` méretet is második paraméterként.

```
// Példa osztályszintű deklarációra
class MyClass {
public:
    // Allokáció
    static void* operator new(size_t size) {
        std::cout << "Egyedi new: " << size << " bájt\n";
        return ::operator new(size); // Globális hívása
    }

    // Deallokáció
    static void operator delete(void* p) {
        std::cout << "Egyedi delete\n";
        ::operator delete(p); // Globális hívása
    }
};
```

31.2 Osztályszintű szabályok (Class-specific)

Ha egy osztályon belül definiáljuk ezeket az operátorokat:

- **Implicit statikusság:** Ezek a tagfüggvények mindig `static`-ok, még akkor is, ha nem írjuk ki eléjük a kulcsszót. Ennek oka, hogy a `new` hívásakor az objektum még nem létezik, a `delete` hívásakor pedig már megszűnt (vagy épp megszűnik).
- **Öröklődés:** A származtatott osztályok öröklik a bázisosztály allokátorait, kivéve, ha felüldefiniálják őket.
- **Tömbös változatok:** A `new[]` és `delete[]` operátorok függetlenek a skalár (egyes) változataiktól, azokat külön kell túlterhelni.

31.3 Globális szabályok (Global scope)

Lehetőség van a globális `operator new` és `operator delete` lecserélésére is:

- **Hatáskör:** Ha globálisan definiáljuk őket, az a teljes programra kihat (beleértve az STL konténereket és a `main`-en kívüli statikus inicializálásokat is).
- **Veszélyek:** A globális csere rendkívül kockázatos; hiba esetén a program összeomolhat vagy memória-szivárgás léphet fel rendszer szinten.
- **Szabványos szignatúra:** Nem lehet namespace-be tenni, a globális névtérben kell lenniük.

31.4 Kivételkezelés és konvenciók

A helyes működés érdekében be kell tartani az alábbi konvenciókat:

- **Sikertelen foglalás:** Ha a `new` nem tud memóriát foglalni, `std::bad_alloc` kivételt kell dobnia (vagy `nullptr`-t visszaadni a `nothrow` változat esetén), nem térhet vissza `NULL`-lal csendben.
- **Végtelen ciklus:** Gyakori minta, hogy a `new` egy ciklusban próbál foglalni, és sikertelenség esetén meghívja a `std::new_handler`-t.
- **Nullptr törlése:** A `delete` operátornak biztonságosan kezelnie kell a `nullptr` bemenetet (ilyenkor nem csinál semmit).
- **Párosítás elve:** Ha túlterheled a `new`-t, kötelező túlterhelni a `delete`-t is! Ha ez elmarad, a memória allokációja az egyedi módon, a felszabadítása viszont az alapértelmezett módon történne, ami undefined behavior-höz vezethet.

31.5 Placement New szabályai

A *placement new* (amely extra paramétereket fogad) speciális eset:

- Csak akkor hívódik hozzá tartozó *placement delete*, ha a konstruktor kivételt dob a létrehozás során.
- Minden egyedi paraméterezésű `operator new`-hoz célszerű definiálni a megfelelő szignatúrájú `operator delete`-t a kivételbiztonság érdekében.

32 Ismertesse az I/O operátorok túlterhelésének szabályait! Írjon példát osztálypéldány kiíratásához!

A C++ nyelvben a saját típusok (osztályok) és a `iostream` könyvtár (pl. `std::cout`, `std::cin`) közötti kommunikációt a biteltoló operátorok (« és ») túlterhelésével valósítjuk meg.

32.1 Alapvető szabályok és elhelyezkedés

Az I/O operátorok túlterhelése eltér a legtöbb operátorétól az operandusok sorrendje miatt.

- **Globális függvényként kell definiálni:**
  - Nem lehetnek az osztály tagfüggvényei.
  - *Indoklás:* A bináris operátoroknál a bal oldali operandus határozza meg a hívást. Kiíratásnál (`std::cout « obj`) a bal oldali operandus az `ostream` típusú objektum, melynek forráskódja (az STL része) nem módosítható.
- **Láncolhatóság (Chaining):**
  - A visszatérési értéknek mindig a kapott stream referenciájának kell lennie.
  - Ez teszi lehetővé a műveletek fűzését: `cout « a « b « c;`.

32.2 A függvények szignatúrája

A szabványos I/O működéshez az alábbi paraméterezést kell követni:

- **Kimenet (Insertion operator «):**
  - **Bal operandus:** `std::ostream&` (nem konstans, mert íráskor változik a belső állapota).
  - **Jobb operandus:** `const T&` (az osztályunk példánya; konstans referenciaként adjuk át a hatékonyság és adatvédelem miatt).
  - **Visszatérés:** `std::ostream&`.
- **Bemenet (Extraction operator »):**
  - **Bal operandus:** `std::istream&`.
  - **Jobb operandus:** `T&` (sima referencia, mivel a függvénynek módosítania kell az objektumot a beolvasott adatokkal).
  - **Visszatérés:** `std::istream&`.

32.3 A „friend” mechanizmus szerepe

Mivel ezek globális (stand-alone) függvények, alapértelmezésben csak a publikus metódusokat érik el.

- Ha a kiíratás/beolvasás `private` adattagokat érint, a függvényt **friend**-ként (barátként) kell deklarálni az osztály belsejében.
- Ha léteznek megfelelő `public` getter/setter metódusok, a friend deklaráció elhagyható (de ez ritkább megoldás).

32.4 Példa: Komplex szám osztály kiíratása

Az alábbi példa bemutatja a deklarációt és az implementációt egy `Complex` osztály esetén.

```cpp
#include <iostream>

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

    // Friend deklaráció az osztályban (a private tagok eléréséhez)
    // Figyelem: ez nem tagfüggvény, csak itt engedélyezzük a hozzáférést
    friend std::ostream& operator<<(std::ostream& os, const Complex& c);
};

// Implementáció (osztályon kívül)
// Paraméterek:
// 1. os: a kimeneti adatfolyam (pl. cout)
// 2. c: a kiírandó objektum
std::ostream& operator<<(std::ostream& os, const Complex& c) {
    // Formázott kiíratás
    os << c.real;
    if (c.imag >= 0) os << "+";
    os << c.imag << "i";

    // Fontos: a stream referenciájának visszaadása a láncoláshoz
    return os;
}

int main() {
    Complex c1(3.5, -2.0);
    Complex c2(1.0, 4.0);

    // Használat láncolva
    // (operator<<(std::cout, c1)) hívódik először, majd az eredményen a c2
    std::cout << "Szamok: " << c1 << " es " << c2 << std::endl;

    return 0;
}
```

33 Ismertesse az „std” névtér „string” osztályát! Adja meg (működés magyarázatával) gyakran használt operátorait és metódusait!

Az `std::string` a C++ szabványos könyvtárának (Standard Template Library - STL) része, amely a szöveges adatok dinamikus, biztonságos és kényelmes kezelését teszi lehetővé. A `<string>` header fájlban található.

33.1 Általános jellemzők

- **Dinamikus memóriakezelés:** Automatikusan foglalja és szabadítja fel a memóriát (RAII elv), a felhasználónak nem kell a `new/delete` párossal törődnie.
- **Skálázhatóság:** Szükség esetén automatikusan növeli a kapacitását.
- **C-kompatibilitás:** Könnyen konvertálható hagyományos C-stílusú (`const char*`) karaktertömbbé.
- **Biztonság:** Mély másolatot (deep copy) készít értékadáskor, elkerülve a mutatók másolásából adódó hibákat.

33.2 Gyakran használt operátorok

A `string` osztály számos operátort túlterhel a természetes használat érdekében:

- **Értékadás (`=`):** Másolatot készít a jobb oldali operandusról.
- **Összefűzés (`+`):** Két stringet vagy egy stringet és egy literált fűz össze, új stringet eredményezve.
- **Hozzáfűzés (`+=`):** A jobb oldali stringet a bal oldali végéhez fűzi (módosítja az eredetit).
- **Indexelés (`[]`):** Elérést biztosít az adott indexű karakterhez (0-tól indexelve). *Megjegyzés:* Nem végez határellenőrzést (gyors, de veszélyes lehet).
- **Összehasonlítás (`==, !=, <, >`):** Lexikografikus (szótári) összehasonlítást végez a stringek tartalma alapján.

33.3 Fontosabb tagfüggvények (metódusok)

- **Méret lekérdezése:**
  - `length()` vagy `size()`: Visszaadja a karakterek számát.
  - `empty()`: Logikai igazat ad, ha a string üres (mérete 0).
- **Hozzáférés és módosítás:**
  - `at(index)`: Mint a `[]`, de határellenőrzést végez (kivételt dob hiba esetén).
  - `clear()`: Törli a string tartalmát, mérete 0 lesz.
  - `push_back(char)`: Egy karaktert fűz a string végére.
- **Keresés és részszöveg:**
  - `c_str()`: Visszaad egy `const char*` mutatót a C-stílusú (null-terminált) változatra. (API hívásokhoz szükséges).
  - `substr(pos, len)`: Részszöveget ad vissza a `pos` indextől kezdve `len` hosszan.
  - `find(str)`: Megkeresi a paraméterként kapott szöveg első előfordulását. Ha nem találja, a visszatérési érték `std::string::npos`.

33.4 Példa a használatra

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

int main() {
    // Konstruktor és értékadás
    std::string s1 = "Hello";
    std::string s2("Vilag");

    // Operátorok: összefűzés és módosítás
    std::string s3 = s1 + " " + s2; // "Hello Vilag"
    s3 += "!";                      // "Hello Vilag!"

    // Metódus: keresés
    // npos: speciális konstans a "találat hiányának" jelzésére
    if (s3.find("Vilag") != std::string::npos) {
        std::cout << "A 'Vilag' szo megtalalhato." << std::endl;
    }

    // Metódus: részszöveg (6. indextől 5 karakter)
    std::string sub = s3.substr(6, 5); // "Vilag"

    // C-kompatibilitás
    const char* c_ptr = s3.c_str();

    std::cout << "Hossz: " << s3.length() << std::endl;

    return 0;
}
```

34 Ismertesse a string-numerikus adat közti konverzióra használt osztályt!

A C++ nyelvben a karakterláncok (stringek) és numerikus típusok (int, double stb.) közötti kétirányú, formázott konverzióra leggyakrabban az `std::stringstream` osztályt használjuk. Ez az osztály a `<sstream>` header fájlban található.

34.1 Az osztály jellemzői és felépítése

A `stringstream` egyesíti a bemeneti és kimeneti adatfolyamok tulajdonságait, de nem konzolra vagy fájlba ír, hanem a memóriában dolgozik.

- **Öröklődés:** Az `iostream` leszármazottja, így ugyanazokkal az operátorokkal («, ») kezelhető, mint a `cin` vagy a `cout`.
- **Belső puffer:** Egy belső `std::string` objektumot kezel pufferként; ebbe írunk bele vagy ebből olvasunk ki.
- **Típusbiztonság:** A C-stílusú `sprintf`-fel ellentétben típusbiztos és nem okoz puffertúlcsordulást.

34.2 Fő metódusok és kezelés

- `str()`:
  - Paraméter nélkül: Visszaadja a belső puffer tartalmát `std::string`-ként (pl. konverzió végeredménye).
  - Paraméterrel (string): Beállítja a belső puffer tartalmát (pl. konverzió kezdete).
- `clear()`:
  - Törli az állapotjelző biteket (pl. EOF, failbit).
  - **Fontos:** Nem törli a tartalmat! Ha újra akarjuk használni a streamet, a tartalmat az `str("")` hívással, a hibaállapotot a `clear()` hívással kell alaphelyzetbe állítani.

34.3 Konverziós irányok

- **Szám → String (Szerializáció):**
  1. Adat beírása a streambe a « operátorral.
  2. Eredmény kinyerése az `.str()` metódussal.
- **String → Szám (Parsolás):**
  1. A string betöltése a streambe (konstruktorban vagy `.str()` hívással).
  2. Adat kiolvasása célváltozóba a » operátorral.
  3. A stream automatikusan kezeli a whitespace karaktereket elválasztóként.

34.4 Mintapélda

```
#include <iostream>
#include <sstream> // Kötelező header
#include <string>

int main() {
    // 1. Konverzió: Szám -> String
```

```cpp
    int szam = 42;
    double lebegopontos = 3.14;

    std::stringstream ss_out;

    // Beírás a streambe (mint a cout-nál)
    ss_out << szam << " " << lebegopontos;

    // Kinyerés stringként
    std::string eredmeny = ss_out.str();
    std::cout << "Stringge alakitva: " << eredmeny << std::endl;

    // ----------------------------------------

    // 2. Konverzió: String -> Szám
    std::string bemenet = "1985 75.5";
    std::stringstream ss_in(bemenet); // Inicializálás stringgel

    int ev;
    float suly;

    // Kiolvasás változókba (mint a cin-nél)
    ss_in >> ev >> suly;

    if (!ss_in.fail()) {
        std::cout << "Ev: " << ev << ", Suly: " << suly << std::endl;
    }

    return 0;
}
```

35 Ismertesse a fájlok kezelésére használt osztályt, gyakran használt metódusait és operátorait!

A C++ nyelvben a fájlkezelés a streameken (adatfolyamokon) keresztül történik, hasonlóan a konzolos kommunikációhoz. A szükséges osztályokat az `<fstream>` header tartalmazza.

35.1 Az alapvető osztályok

A fájlkezelés iránya határozza meg, melyik osztályt használjuk:

- **`std::ofstream` (Output File Stream):** Fájlba írásra szolgál. Ha a fájl nem létezik, létrehozza.
- **`std::ifstream` (Input File Stream):** Fájlból való olvasásra szolgál.
- **`std::fstream` (File Stream):** Kétirányú kommunikációt (írást és olvasást is) lehetővé tesz.

35.2 Megnyitás és fájlmódok

A fájlokat megnyithatjuk a konstruktorban vagy az `open()` metódussal. A második paraméter határozza meg a megnyitás módját (ezek kombinálhatók a `|` operátorral):

- **`std::ios::in`:** Megnyitás olvasásra (alapértelmezett `ifstream`-nél).
- **`std::ios::out`:** Megnyitás írásra (alapértelmezett `ofstream`-nél). Felülírja a fájlt!
- **`std::ios::app`:** Hozzáfűzés (Append). A meglévő tartalom megmarad, az írás a végére kerül.
- **`std::ios::binary`:** Bináris mód (pl. képek, struktúrák mentésekor), kikapcsolja a szöveges konverziókat.

35.3 Fontos tagfüggvények (Metódusok)

- **`open(filename, mode)`:** Hozzárendeli a streamet egy fizikai fájlhoz.
- **`is_open()`:** Logikai értékkel tér vissza: sikerült-e a fájl megnyitása? (Mindig ellenőrizni kell!).
- **`close()`:** Bezárja a fájlt és menti a pufferek tartalmát. (A destruktor is meghívja, de ajánlott explicit módon használni).
- **`eof()`:** (End Of File) Igazat ad, ha elértük a fájl végét olvasáskor.
- **`getline(stream, string)`:** Globális segédfüggvény, amely egy teljes sort olvas be a fájlból (a szóközt is beleértve), amíg sortörést nem talál.

35.4 Operátorok

Mivel az osztályok az `iostream`-ből származnak, az operátorok megegyeznek a konzolos I/O-val:

- **`<<` (Insertion):** Adat írása a fájlba (formázott szövegként).
- **`>>` (Extraction):** Adat olvasása a fájlból (whitespace karaktereknél megáll).

35.5 Példa: Írás és olvasás

```cpp
#include <iostream>
#include <fstream>
#include <string>

int main() {
    // 1. Írás fájlba
    std::ofstream kimenet("adatok.txt"); // Létrehozás és megnyitás
    if (kimenet.is_open()) {
        kimenet << "Elso sor" << std::endl;
        kimenet << 123 << std::endl;
        kimenet.close(); // Lezárás
    }

    // 2. Olvasás fájlból
    std::ifstream bemenet("adatok.txt");
    std::string sor;

    if (bemenet.is_open()) {
        // Soronkénti beolvasás while ciklussal
        // A getline visszatérési értéke maga a stream, ami false-t ad hiba/EOF esetén
        while (std::getline(bemenet, sor)) {
            std::cout << "Beolvasva: " << sor << std::endl;
        }
        bemenet.close();
    } else {
        std::cerr << "Hiba a fajl megnyitasakor!" << std::endl;
    }

    return 0;
}
```

36 Ismertesse példával a „kompozíció” elvet osztályok egymásba ágyazására!

A kompozíció (összetétel) az objektumorientált programozás egyik alapvető építőköve, amely a „tartalmazás” (vagy „has-a”, azaz „van neki”) kapcsolatot valósítja meg két osztály között.

36.1 A kompozíció elvei

- **„Has-a” kapcsolat:** Az öröklődéssel ellentétben (ami „is-a” típusú), itt az egyik objektum birtokolja a másikat (pl. a Számítógépnek *van* Processzora).
- **Szoros csatolás (Strong association):** A tartalmazott objektum (rész) élettartama függ a tartalmazó objektum (egész) élettartamától. Ha az „egész” megszűnik, a „rész” is megsemmisül.
- **Újrafelhasználhatóság:** Lehetővé teszi bonyolult objektumok felépítését egyszerűbb, már meglévő osztályokból.

36.2 Megvalósítás C++ nyelven

Technikailag a kompozíciót úgy valósítjuk meg, hogy egy osztály típusú változót adattagként deklarálunk egy másik osztályban.

- **Adattag:** Az objektumot érték szerint tároljuk (nem mutatóként), így a memóriakezelést a fordító automatikusan végzi.
- **Member Initializer List:** A tartalmazott objektum konstruktorát a tartalmazó osztály konstruktorának inicializáló listájában kell meghívni. Ez kritikus fontosságú, ha a belső objektumnak nincs paraméter nélküli (default) konstruktora.

36.3 Példa: Számítógép és Processzor

Az alábbi példában a `Szamitogep` osztály kompozícióval tartalmazza a `CPU` osztályt. A CPU inicializálása a Számítógép létrehozásakor történik.

```cpp
#include <iostream>

// A "rész" osztály
class CPU {
    int frekvencia;
public:
    // Nincs default konstruktor, paraméter kötelező
    CPU(int freq) : frekvencia(freq) {
        std::cout << "CPU beepitve: " << frekvencia << " MHz" << std::endl;
    }

    void dolgozik() {
        std::cout << "CPU szamol..." << std::endl;
    }
};

// Az "egész" osztály
class Szamitogep {
private:
    // Kompozíció: A CPU a Szamitogep adattagja
```

```
21      CPU processzor;
22
23  public:
24      // Konstruktor
25      // A "processzor" adattagot az inicializáló listán KELL beállítani,
26      // mielőtt a konstruktor törzse lefutna.
27      Szamitogep(int freq) : processzor(freq) {
28          std::cout << "Szamitogep bekapcsolva." << std::endl;
29      }
30
31      void futtat() {
32          // A feladatot delegáljuk a belső objektumnak
33          processzor.dolgozik();
34      }
35  };
36  // Destruktor lefutásakor: először a Szamitogep szűnik meg,
37  // majd automatikusan a processzor is.
38
39  int main() {
40      Szamitogep pc(3200);
41      pc.futtat();
42      return 0;
43  }
```

37 Ismertesse az „aggregáció” elvet osztályok egymásba ágyazására!

Az aggregáció az objektumorientált programozásban a „tartalmazás” (association) egy speciális formája. Hasonló a kompozícióhoz (mindkettő „has-a” kapcsolat), de a kapcsolat erőssége és az objektumok élettartama alapvetően eltér.

37.1 Az aggregáció jellemzői

- **Laza kapcsolat (Weak association):** A tartalmazó objektum (Egész) és a tartalmazott objektum (Rész) kapcsolata nem kizárólagos.
- **Független élettartam:** A „rész” objektum létezhet az „egész” nélkül is. Ha a tartalmazó objektum megszűnik, a benne hivatkozott objektum **nem** semmisül meg automatikusan.
- **Megosztott birtoklás:** Ugyanaz a „rész” objektum egyszerre tartozhat több különböző „egészhez” is (pl. egy Tanár taníthat több Osztályban is).

37.2 Megvalósítás C++ nyelven

Technikailag az aggregációt mutatók (pointerek) vagy referenciák használatával valósítjuk meg, nem érték szerinti tárolással.

- **Adattag:** Az osztály egy mutatót (`T*`) vagy referenciát (`T&`) tárol a másik osztályra.
- **Konstruktor:** A külső objektumot általában paraméterként kapja meg a konstruktor vagy egy setter metódus (nem ő hozza létre `new`-val).
- **Destruktor:** A tartalmazó objektum destruktora **nem** hívja meg a `delete`-t a hivatkozott objektumra (hiszen nem ő a tulajdonosa).

37.3 Példa: Autó és Sofőr

Az alábbi példában az Autó aggregálja a Sofőrt. A Sofőr létezik az Autó előtt is, és az Autó megsemmisülése után is tovább él.

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

// A "rész" osztály (ami független is lehet)
class Sofor {
public:
    std::string nev;
    Sofor(std::string n) : nev(n) {}

    void vezet() {
        std::cout << nev << " vezeti az autot." << std::endl;
    }
};

// Az "egész" osztály
class Auto {
private:
    // Aggregáció: Mutatót tárolunk, nem magát az objektumot!
    Sofor* sofor;
```

```
20
21 public:
22     // Kezdetben lehet, hogy nincs sofőr (nullptr)
23     Auto() : sofor(nullptr) {}
24
25     // Sofőr hozzárendelése (nem itt hozzuk létre!)
26     void setSofor(Sofor* s) {
27         sofor = s;
28     }
29
30     void indul() {
31         if (sofor != nullptr) {
32             sofor->vezet();
33         } else {
34             std::cout << "Nincs sofor, az auto nem indul." << std::endl;
35         }
36     }
37
38     // Destruktor: NEM töröljük a sofőrt, mert nem mi birtokoljuk!
39     ~Auto() {
40         std::cout << "Az auto megsemmisult." << std::endl;
41     }
42 };
43
44 int main() {
45     // 1. Létrehozzuk a sofőrt (független objektum)
46     Sofor* janos = new Sofor("Janos");
47
48     {
49         // 2. Létrehozunk egy autót egy belső blokkban
50         Auto taxi;
51         taxi.setSofor(janos); // Összekapcsoljuk őket
52         taxi.indul();
53
54     } // 3. Itt az 'Auto' (taxi) megsemmisül, lefut a destruktora
55
56     // 4. A 'Sofor' (janos) MÉG MINDIG LÉTEZIK és használható
57     std::cout << "Janos meg mindig megvan: " << janos->nev << std::endl;
58
59     // A tulajdonos felelőssége a törlés
60     delete janos;
61
62     return 0;
63 }
```

38 Ismertesse az „öröklődés” elvet osztályok egymásba ágyazására! Mi az öröklődés szintaktikája a C++-ban?

Az öröklődés (inheritance) az objektumorientált programozás egyik legfontosabb pillére, amely lehetővé teszi, hogy egy meglévő osztályból (ős) új osztályt (utód) hozzunk létre, átvéve annak tulajdonságait és viselkedését.

38.1 Az elv és a kapcsolat típusa

- **„Is-a” kapcsolat:** Az öröklődés a „van egy” (típusú) kapcsolatot valósítja meg. Például: a „Kutya” *egy* „Állat”. (Eltér a kompozíció „has-a” kapcsolatától).
- **Hierarchia:**
  - **Bázisosztály (Base class):** Az ős, amely az általános tulajdonságokat tartalmazza.
  - **Származtatott osztály (Derived class):** Az utód, amely örökli az őst, és specifikus tulajdonságokkal/metódusokkal egészíti ki vagy módosítja azt.
- **Kódújrafelhasználás:** A közös logikát elég egyszer, az ősben megírni, az utódok automatikusan megkapják.

38.2 Szintaktika C++ nyelven

Az öröklést az osztály definíciójakor, a név után kettősponttal adjuk meg.

```
1 class Szarmaztatott : [hozzaferes] BazisOsztaly {
2     // A származtatott osztály törzse
3 };
```

Ahol a `[hozzaferes]` a származtatás módja lehet: `public`, `protected` vagy `private`.

38.3 Származtatási módok (Láthatóság)

A származtatási mód határozza meg, hogy az örökölt tagok hogyan látszanak az utódban:

- **`public` (Leggyakoribb):**
  - Az ős `public` tagjai `public` maradnak.
  - Az ős `protected` tagjai `protected` maradnak.
  - Az ős `private` tagjai nem érhetők el közvetlenül.
  - *Jelentése:* Az utód teljes mértékben helyettesítheti az őst (interfész öröklés).
- **`protected`:**
  - Az ős `public` és `protected` tagjai `protected` elérésűvé válnak az utódban.
- **`private`:**
  - Minden örökölt tag `private` lesz az utódban. Ez inkább implementációs öröklés (hasonlít a kompozícióhoz).

38.4 Konstruktorok és Destruktorok sorrendje

Az öröklődés során a létrejövés és megszűnés sorrendje szigorúan kötött:

1. **Létrehozás (Konstruktor):** Először a **Bázisosztály** konstruktora fut le (hogy az alapok készen álljanak), utána a **Származtatott** osztályé.
2. **Megszűnés (Destruktor):** Fordított sorrendben történik. Először a **Származtatott** osztály takarít, végül a **Bázisosztály**.

Megjegyzés: Ha az ősnek van paraméteres konstruktora, azt az utód *Member Initializer List*-jében (taginicializáló lista) kell meghívni.

38.5 Példa

```cpp
#include <iostream>

// Bázisosztály
class Allat {
protected:
    int labakSzama; // Protected: utód látja, külvilág nem

public:
    Allat(int lab) : labakSzama(lab) {
        std::cout << "Allat letrejott." << std::endl;
    }

    void eszik() {
        std::cout << "Az allat eszik." << std::endl;
    }
};

// Származtatott osztály (Public öröklés)
class Kutya : public Allat {
public:
    // Az ős konstruktorát hívjuk az inicializáló listán
    Kutya() : Allat(4) {
        std::cout << "Kutya letrejott." << std::endl;
    }

    void ugat() {
        // Hozzáférünk a protected taghoz
        std::cout << "Vau! Labaim szama: " << labakSzama << std::endl;
    }
};

int main() {
    Kutya bodri;      // 1. Allat ctor, 2. Kutya ctor
    bodri.eszik();    // Örökölt metódus
    bodri.ugat();     // Saját metódus
    return 0;         // Destruktorok: 1. Kutya dtor, 2. Allat dtor
}
```

39 Csoportosítsa az osztályban található elemeket öröklődési szempontból: mely elemek öröklődnek, és mely elemek nem öröklődnek?

A C++ öröklődési modelljében különbséget teszünk azon elemek között, amelyek automatikusan az utódosztály részévé válnak, és azok között, amelyek szorosan a bázisosztály identitásához (létrehozás, másolás, megszűnés) kötődnek, ezért nem öröklődnek.

39.1 Örökölt elemek

Az alábbi elemeket a származtatott osztály megkapja (láthatóságuk a hozzáférési módosítóktól függ, de fizikailag vagy logikailag jelen vannak):

- **Adattagok (Member variables):**
  - Minden statikus (`static`) és nem statikus adattag öröklődik.
  - *Megjegyzés:* A `private` tagok is öröklődnek (lefoglalásra kerülnek a memóriában az utód objektumában), de a kódban közvetlenül nem érhetők el az utódból.
- **Tagfüggvények (Member functions):**
  - Minden statikus és nem statikus metódus.
  - Virtuális függvények (ezek felüldefiniálhatók az utódban).
- **Belső típusok:**
  - Az osztályon belül definiált `typedef`-ek, `enum`-ok, `struct`-ok vagy osztályok (amennyiben a hozzáférés ezt engedi).

39.2 Nem örökölt elemek

Ezeket az elemeket a fordító minden osztályhoz egyedileg rendeli (vagy generálja le), nem vehetők át automatikusan a szülőből:

- **Konstruktorok:**
  - Az alapértelmezett, paraméteres, másoló és mozgató (move) konstruktorok nem öröklődnek.
  - *Kivétel C++11 óta:* Az `using Base::Base;` utasítással explicit módon „behúzhatók” (inheriting constructors), de alapértelmezésben nem járnak.
- **Destruktor:**
  - Minden osztálynak saját destruktora van. (Bár az utód destruktora automatikusan meghívja az ősét, de nem „örökli” azt).
- **Értékadó operátor (`operator=`):**
  - A fordító minden osztályhoz saját `operator=`-t generál, ha nincs megírva. Ez elrejti az ős értékadó operátorát.
- **Friend (Barát) relációk:**
  - „Az apám barátja nem az én barátom.” A barátság nem öröklődik és nem tranzitív.

Elem típusa	Öröklődik?
Adattagok (int x, static int y)	Igen
Tagfüggvények (void func())	Igen
Konstruktorok	Nem
Destruktor	Nem
Assignment operator (=)	Nem
Friend deklarációk	Nem

39.3 Összefoglaló táblázat és példa

```
class Base {
public:
    int x;              // ÖRÖKLŐDIK
    void f() {}         // ÖRÖKLŐDIK

    Base() {}           // NEM öröklődik (de hívódik)
    ~Base() {}          // NEM öröklődik (de hívódik)

    void operator=(const Base&) {} // NEM öröklődik

    friend void barat(); // NEM öröklődik
private:
    int y;              // ÖRÖKLŐDIK (memóriában ott van), de nem látszik
};
```

40 Ismertesse öröklődés során a leszármazottban található konstruktor paraméterezésének és hívásának szabályait, tekintettel az ősben levő privát adattagokra!

Az objektumorientált programozásban az öröklődés során a származtatott osztály (utód) objektuma magában foglalja az alaposztály (ős) adattagjait is. A helyes inicializálás kulcsa a konstruktorok láncolása.

40.1 A probléma: Privát adattagok elérése

- **Láthatósági korlát:** Az ősosztály `private` adattagjai fizikailag jelen vannak az utód memóriaképében, de az utódosztály kódjából közvetlenül nem érhetők el (sem olvasásra, sem írásra).
- **Közvetlen értékadás tilalma:** Az utód konstruktorának törzsében nem írhatjuk le, hogy `os_privat_adat = ertek;`, mert ez hozzáférési hibát (access violation) okoz.
- **Megoldás:** Az inicializálás felelősségét át kell adni az ősosztálynak, aki hozzáfér a saját privát adataihoz.

40.2 Konstruktor hívási szabályok

Az utódosztály konstruktorának „közvetítőként” kell viselkednie:

- **Taginicializáló lista (Member Initializer List):** Az ős konstruktorát **kizárólag** az inicializáló listán (a kettőspont után, de a kapcsos zárójel előtt) lehet és kell meghívni.
- **Paraméterátadás:** Az utód konstruktora paraméterként bekéri az összes adatot (a sajátjaihoz és az őséihez tartozókat is), majd a megfelelőket továbbpasszolja az ős konstruktorának.
- **Végrehajtási sorrend:**
  1. Először lefut az **ős** konstruktora (inicializálja a privát adattagokat).
  2. Ezután inicializálódnak az **utód** saját adattagjai.
  3. Végül lefut az **utód** konstruktorának törzse.

40.3 Kötelezőség és Default konstruktor

- **Ha van Default (paraméter nélküli) konstruktor az ősben:** Nem kötelező explicit módon hívni az őst az inicializáló listán; a fordító automatikusan meghívja a paraméter nélkülit.
- **Ha NINCS Default konstruktor az ősben:** Az utódnak **kötelező** explicit módon meghívnia az ős valamelyik paraméteres konstruktorát. Ennek hiányában a kód nem fordul le ("no default constructor available").

40.4 Példa

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

// Ősosztály
class Jarmu {
private:
    int loero; // Privát: az Auto nem látja közvetlenül!

public:
    // Paraméteres konstruktor (Nincs default!)
```

```
11      Jarmu(int hp) : loero(hp) {
12          std::cout << "Jarmu init: " << loero << " HP" << std::endl;
13      }
14  };
15
16  // Származtatott osztály
17  class Auto : public Jarmu {
18  private:
19      std::string marka; // Saját adattag
20
21  public:
22      // A konstruktor paraméterben kapja meg a lóerőt (ősnek) és a márkát (magának)
23      Auto(int hp, std::string m)
24          : Jarmu(hp),  // 1. Továbbítjuk az adatot az ősnek (KÖTELEZŐ itt!)
25            marka(m)    // 2. Inicializáljuk a saját adatot
26      {
27          // 3. Itt a 'loero' már be van állítva, de
28          // loero = 100; // HIBA lenne, mert privát
29          std::cout << "Auto kesz." << std::endl;
30      }
31  };
```

41 Ismertesse az ősben található osztálytagok elérésének módosítását private és protected öröklődés során!

C++-ban az öröklődés típusa (hozzáférési módosítója) szabályozza, hogy az ősosztály tagjai milyen láthatósággal jelenjenek meg a származtatott osztályban. Általános szabály, hogy az öröklés soha nem tágítja, csak szűkítheti (vagy szinten tarthatja) a láthatóságot.

41.1 Az alapvető mechanizmus

A végső elérési szintet a „legszigorúbb szabály" elve határozza meg:

$$\text{Új láthatóság} = \max(\text{Eredeti láthatóság}, \text{Öröklés típusa})$$

Ahol a szigorúsági sorrend: `private` > `protected` > `public`. Az ős `private` tagjai soha nem érhetők el közvetlenül a származtatott osztályból, függetlenül az öröklés típusától.

41.2 Protected (Védett) öröklődés

Ha `class Derived : protected Base` formában származtatunk:

- **Hatása:**
  - Az ős **public** tagjai → **protected** tagokká válnak az utódban.
  - Az ős **protected** tagjai → **protected** tagok maradnak.
  - Az ős **private** tagjai → elérhetetlenek maradnak.
- **Következmény:**
  - A külvilág (pl. `main` függvény) számára az összes örökölt tag elérhetetlenné válik (mivel védettek lettek).
  - A további leszármazottak (az unokák) viszont még hozzáférhetnek ezekhez a tagokhoz (mivel protected státuszúak).

41.3 Private (Privát) öröklődés

Ha `class Derived : private Base` formában származtatunk (vagy elhagyjuk a kulcsszót `class` esetén):

- **Hatása:**
  - Az ős **public** tagjai → **private** tagokká válnak az utódban.
  - Az ős **protected** tagjai → **private** tagokká válnak az utódban.
  - Az ős **private** tagjai → elérhetetlenek maradnak.
- **Következmény:**
  - A külvilág számára minden el van rejtve.
  - A további leszármazottak (unokák) már semmit sem látnak az ősosztályból (mivel itt a lánc megszakad a priváttá tétel miatt).
  - Ez gyakorlatilag implementációs öröklés (hasonló a kompozícióhoz).

Ős tagja	Public öröklés	Protected öröklés	Private öröklés
`public`	`public`	`protected`	`private`
`protected`	`protected`	`protected`	`private`
`private`	*Elérhetetlen*	*Elérhetetlen*	*Elérhetetlen*

41.4 Összefoglaló táblázat

41.5 Láthatóság visszaállítása (Using declaration)

Privát vagy védett öröklés esetén is visszaállítható egyes tagok láthatósága a `public` szintre a `using` kulcsszóval.

```
class Base {
public:
    void fgv() {}
    int adat;
};

class Derived : private Base {
public:
    // Kivétel: ezt az egy tagot publikussá tesszük
    using Base::fgv;
    // Az 'adat' továbbra is private marad
};
```

41.6 Példa a korlátozásokra

```
class Base {
public:    int pub;
protected: int prot;
private:   int priv;
};

// 1. Protected öröklés
class ProtDerived : protected Base {
    void teszt() {
        pub = 1;  // OK (itt protected)
        prot = 2; // OK (itt protected)
        // priv = 3; // HIBA: ős privátja nem érhető el
    }
};

// 2. Private öröklés
class PrivDerived : private Base {
    void teszt() {
        pub = 1;  // OK (itt private)
        prot = 2; // OK (itt private)
    }
};

class Unoka : public PrivDerived {
    void teszt() {
```

```
26          // pub = 1; // HIBA! A PrivDerived-ben ez már private lett!
27      }
28  };
29
30  int main() {
31      ProtDerived pd;
32      // pd.pub = 1; // HIBA! Kívülről protected, nem látszik.
33      return 0;
34  }
```

42 Ismertesse az osztálytagok elérését ősosztály típusú pointerrel! Mi a „korai kötés” működése és problémája?

A C++ polimorfizmusának alapja, hogy egy ősosztály típusú mutató (`Base*`) képes tárolni egy leszármazott osztály (`Derived`) példányának címét. Ennek kezelése azonban szigorú szabályokhoz kötött.

42.1 Az ősosztály típusú mutató viselkedése

Amikor egy `Base*` típusú mutatón keresztül érünk el egy `Derived` objektumot:

- **Láthatósági korlát (Interface Slicing):** A mutatón keresztül kizárólag azok a tagok (változók és függvények) érhetők el, amelyek az **ősosztályban** deklarálva vannak.
- **Leszármazott tagjai:** A leszármazottban hozzáadott *új* adattagok és metódusok a fordító számára „láthatatlanok” maradnak ezen a mutatón keresztül (bár a memóriában ott vannak).
- **Cél:** Ez biztosítja a típusbiztonságot; a fordító garantálja, hogy amit meghívunk, az biztosan létezik az ős interfészében.

42.2 A korai kötés (Early / Static Binding)

A korai kötés a C++ alapértelmezett működési módja (ha nem használunk `virtual` kulcsszót).

- **Működése:** A fordítóprogram **fordítási időben** (compile time) dönti el, hogy egy függvényhívás melyik memóriacímen lévő kódra ugorjon.
- **Döntés alapja:** A döntés kizárólag a mutató **statikus típusán** alapul (annak a típusnak, aminek deklaráltuk a változót), nem pedig azon, hogy futásidőben milyen objektumra mutat.
- **Sebesség:** Ez a leggyorsabb hívási mód, mivel nincs futásidejű adminisztráció (vtable keresés).

42.3 A probléma: A polimorfizmus hiánya

Ha a leszármazott osztályban felüldefiniálunk (override) egy metódust, de az ősben nem jelöltük `virtual`-ként, a korai kötés logikai hibához vezethet:

- **A jelenség:** Hiába tartalmazza a memória a `Derived` objektumot a saját, módosított metódusával, a `Base*` mutató miatt a fordító az ősosztály metódusát ("régi kód") köti be.
- **Eredmény:** Az objektum nem az elvárt (specifikus), hanem az általános (ős) viselkedést mutatja. Ez ellehetetleníti a valódi polimorf működést.

42.4 Példa a hibás működésre

Az alábbi példában a `koszon()` függvényt felüldefiniáltuk, de a `virtual` kulcsszó hiánya miatt a korai kötés érvényesül.

```cpp
#include <iostream>

class Ember {
public:
    // Nincs 'virtual' -> Korai kötés
    void koszon() {
        std::cout << "Szia, ember vagyok!" << std::endl;
    }
};
```

```cpp

class Diak : public Ember {
public:
    // Hiába definiáljuk felül
    void koszon() {
        std::cout << "Jo napot, diak vagyok!" << std::endl;
    }

    void tanul() { std::cout << "Tanulok..." << std::endl; }
};

int main() {
    Diak* d = new Diak();

    // 1. Eset: Diák mutató
    d->koszon(); // Kiírja: "Jo napot, diak vagyok!" (Helyes)
    d->tanul();  // Eléri az új metódust is

    // 2. Eset: Ős (Ember) mutató
    Ember* e = d; // Ugyanarra a Diák objektumra mutat!

    // A PROBLÉMA ITT LÁTHATÓ:
    // A fordító csak azt látja, hogy 'e' egy Ember*.
    // Ezért az Ember::koszon() függvényt hívja meg fixen.
    e->koszon(); // Kiírja: "Szia, ember vagyok!" (Helytelen/Nem elvárt)

    // e->tanul(); // HIBA: Az 'Ember' osztályban nincs 'tanul'

    delete d;
    return 0;
}
```

43 Ismertesse a C++-ban található „többszörös öröklődés” elvet! Az ismertetést ábrával és program-részlettel illusztrálja!

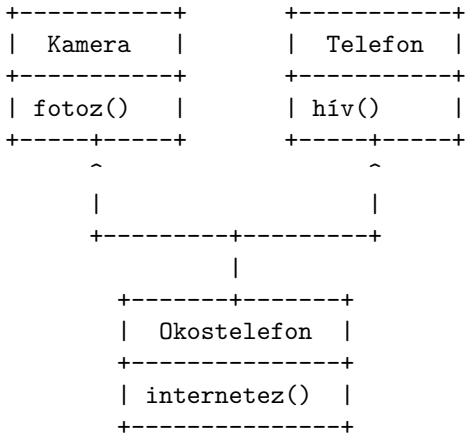
A többszörös öröklődés (Multiple Inheritance) azt a mechanizmust jelenti, amikor egy származtatott osztálynak **egynél több közvetlen ősosztálya** van. Ekkor az utód az összes felsorolt ős tulajdonságait (adattagjait és metódusait) egyesíti magában.

43.1 Alapvető szabályok

- **Szintaxis:** Az osztály definíciójában vesszővel elválasztva soroljuk fel az ősöket, mindegyiknél külön megadva a láthatóságot (pl. `public`).
- **Konstruktorok sorrendje:** Az ősök konstruktorai a *deklarációban felsorolt sorrendben* futnak le (nem az inicializáló lista sorrendje számít!).
- **Destruktorok sorrendje:** A konstruktorokkal ellentétes (fordított) sorrendben.

43.2 Strukturális Ábra

Az alábbi ábra a *Kamera* és *Telefon* osztályok egyesítését mutatja egy *Okostelefon* osztályban.



43.3 Problémák és megoldások

- **Névütközés (Ambiguity):** Ha két ősosztályban azonos nevű függvény van, a fordító nem tudja, melyiket hívja.
  - *Megoldás:* Scope feloldó operátor használata: `obj.Os1::fgv()`.
- **Rombusz-probléma (Diamond Problem):** Ha két ősosztálynak (B és C) van egy közös őse (A), akkor a végső utódban (D) az ’A’ adattagjai duplán jelennének meg.
  - *Megoldás:* **Virtual inheritance** (`virtual public Base`). Így csak egyetlen példány jön létre a közös ősből.

43.4 Példa program

Az alábbi példa bemutatja két független osztály egyesítését és a névütközés feloldását.

```
1 #include <iostream>
2
3 // 1. Ősosztály
```

```cpp
class Kamera {
public:
    void fotoz() {
        std::cout << "Katt!" << std::endl;
    }

    void bekapcsol() { // Névütközés forrása lesz
        std::cout << "Kamera be." << std::endl;
    }
};

// 2. Ősosztály
class Telefon {
public:
    void hiv() {
        std::cout << "Hivas inditasa..." << std::endl;
    }

    void bekapcsol() { // Névütközés forrása lesz
        std::cout << "Telefon be." << std::endl;
    }
};

// Származtatott osztály: mindkettőből örököl
class Okostelefon : public Kamera, public Telefon {
public:
    void internetez() {
        std::cout << "Bongeszes..." << std::endl;
    }
};

int main() {
    Okostelefon mobil;

    // Egyedi funkciók elérése gond nélkül
    mobil.fotoz();
    mobil.hiv();
    mobil.internetez();

    // Névütközés kezelése
    // mobil.bekapcsol(); // HIBA: "ambiguous" (kétértelmű)

    // Helyes hívás scope feloldással:
    mobil.Kamera::bekapcsol();
    mobil.Telefon::bekapcsol();

    return 0;
}
```

44 Ismertesse példával a „virtuális metódus” elv működését! Mit tartalmaz a VMT (vftable) táblázat? A leszármazottban is ugyanazt a szintaktikát kell használni a virtuális metódus felülírásakor?

A virtuális metódusok teszik lehetővé a valódi polimorfizmust (késői kötést) C++-ban. Segítségükkel az ősosztályra mutató pointeren keresztül is a ténylegesen létrehozott (leszármazott) objektum megfelelő metódusa hívódik meg.

44.1 A működési elv: Késői kötés (Late Binding)

A `virtual` kulcsszó használatakor a fordító nem fordítási időben (statikusan) dönti el a függvényhívás címét, hanem futásidőre halasztja azt.

- **Döntés alapja:** A hívás nem a mutató típusától függ (pl. `Base*`), hanem attól, hogy a memóriában ténylegesen milyen típusú objektum van (pl. `Derived`).
- **Eredmény:** Ha van egy `Base* p = new Derived();` pointerünk, akkor a `p->virtualFuggveny()` a `Derived` implementációját fogja futtatni.

44.2 A VMT (Virtual Method Table) felépítése

A mechanizmus hátterében egy lookup tábla, a **VMT** (vagy vtable/vftable) áll.

- **Mit tartalmaz a táblázat?**
  - Függvénycímeket (pointereket).
  - Minden olyan osztályhoz készül egy statikus tábla, amelynek van legalább egy virtuális függvénye.
  - A táblázat sorai az adott osztályhoz érvényes virtuális függvények memóriacímeit tartalmazzák (vagy a sajátját, vagy ha nem írta felül, akkor az örökölt ősét).
- **A vptr (Virtual Pointer):**
  - Minden objektum, amely virtuális metódusokkal rendelkező osztályból származik, tartalmaz egy rejtett mutatót (`vptr`).
  - Ez a mutató az objektum létrehozásakor (a konstruktorban) beállítódik a saját osztálya VMT-jére.
  - Híváskor a program: `objektum` → `vptr` → `VMT` → `helyes függvénycím`.

44.3 Szintaktika a leszármazottban

A felülírás (overriding) szabályai szigorúak, de a szintaxis rugalmas:

- **Szignatúra egyezése:** A visszatérési értéknek, a névnek és a paraméterlistának **pontosan** meg kell egyeznie az ősben lévővel.
- **virtual kulcsszó:** A leszármazottban **nem kötelező** kiírni a `virtual` szót (ha az ősben az volt, akkor automatikusan öröklődik a tulajdonság), de az olvashatóság miatt ajánlott.
- **override kulcsszó (C++11):** Erősen ajánlott a függvény deklarációja után írni az `override` szót. Ez biztosítja, hogy a fordító hibát dobjon, ha véletlenül elírtuk a függvény nevét vagy paramétereit, és emiatt nem jött létre felülírás.

44.4 Példa

```cpp
#include <iostream>

class Alakzat {
public:
    // A VMT bejegyzés létrehozása
    virtual void rajzol() {
        std::cout << "Valamilyen alakzat" << std::endl;
    }
};

class Kor : public Alakzat {
public:
    // Felülírás (Override)
    // A 'virtual' elhagyható lenne, de az 'override' segít a hibaszűrésben
    void rajzol() override {
        std::cout << "O egy Kor" << std::endl;
    }
};

int main() {
    // Ős típusú mutató, de Leszármazott objektum
    Alakzat* a = new Kor();

    // Működés lépései:
    // 1. 'a' pointer a memóriában lévő objektumra mutat.
    // 2. Kiolvassa az objektum rejtett 'vptr'-ét.
    // 3. A vptr a 'Kor' VMT-jére mutat.
    // 4. A VMT-ből kiveszi a 'Kor::rajzol' címét.
    a->rajzol(); // Kimenet: "O egy Kor"

    delete a;
    return 0;
}
```

45 Ismertesse ábrával a „közvetlen bázisosztály” és a „közvetett bázisosztály” fogalmakat!

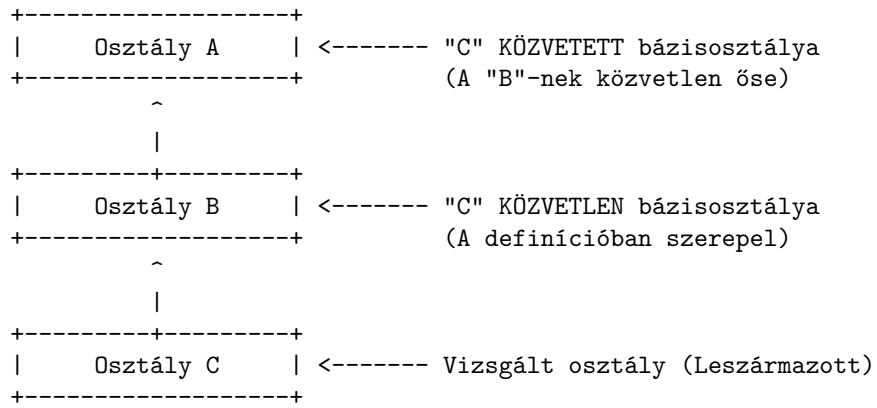
Az objektumorientált programozásban az öröklődési hierarchia mélysége alapján különböztetjük meg az ősöket. Ez a megkülönböztetés fontos a névütközések feloldása, a konverziók és a konstruktorhívások szempontjából.

45.1 Fogalmak definíciója

- **Közvetlen bázisosztály (Direct Base Class):** Az az osztály, amelyből az adott osztályt specifikusan származtattuk. Ez az osztály szerepel a leszármazott osztály definíciójában a kettőspont után.
- **Közvetett bázisosztály (Indirect Base Class):** Az öröklődési láncban feljebb (távolabb) elhelyezkedő ősök (pl. a „nagyszülő”). Ezek tulajdonságait a leszármazott a köztes osztályokon keresztül, tranzitív módon örökli.

45.2 Strukturális Ábra

Az alábbi diagram a **„C” osztály szemszögéből** mutatja be a relációkat egy többszintű öröklődés (Multilevel Inheritance) esetén.



45.3 Programrészlet és Szintaktika

```cpp
// 1. A legfelső szint
class A {
public:
    int x;
};

// 2. Köztes szint
// Itt: 'A' a 'B' osztály KÖZVETLEN bázisosztálya
class B : public A {
public:
    int y;
};

// 3. Alsó szint
// Itt: 'B' a 'C' osztály KÖZVETLEN bázisosztálya
// Itt: 'A' a 'C' osztály KÖZVETETT bázisosztálya
class C : public B {
```

```
18 public:
19     void teszt() {
20         x = 10; // Eléri a közvetett ős tagját is (ha public/protected)
21         y = 20; // Eléri a közvetlen ős tagját is
22     }
23 };
```

45.4 Fontos szabályok

- **Konstruktor hívás:** A leszármazott osztály konstruktorának inicializáló listájában (`C() : ...`) **kizárólag a közvetlen bázisosztály** (`B`) konstruktora hívható meg. A közvetett ős (`A`) inicializálása a köztes osztály (`B`) felelőssége.
- **Virtuális öröklés kivétele:** Virtuális öröklés esetén (`virtual inheritance`) a „legalsó” leszármazott felelős a virtuális közvetett ős inicializálásáért.

46 Ismertesse ábrával a virtuális öröklődés szükségességét előidéző szituációt!

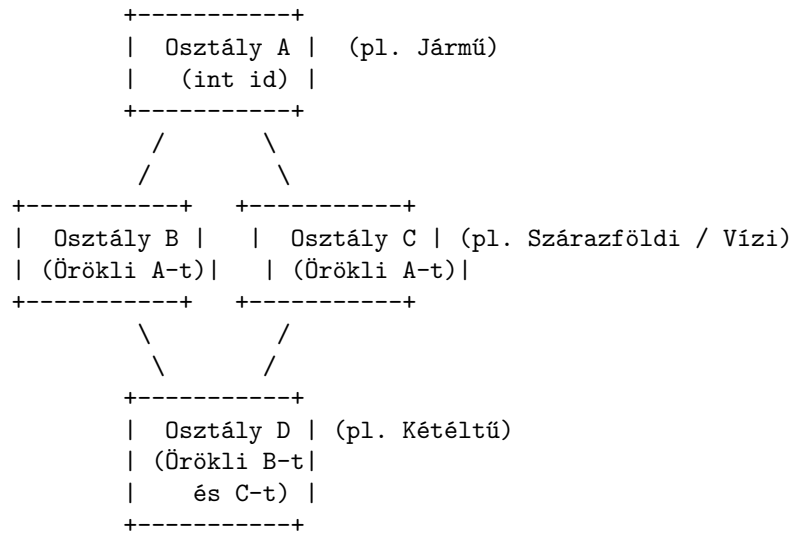
A virtuális öröklődés szükségessége a **többszörös öröklődés** egy speciális esetében, az úgynevezett **Rombusz-probléma** (Diamond Problem) során merül fel.

46.1 A probléma leírása

Ha egy osztálynak (D) két olyan közvetlen őse van (B és C), amelyek ugyanabból a közös ősosztályból (A) származnak, akkor a normál öröklődés során duplikáció és kétértelműség lép fel.

- **Adatduplikáció:** A végső leszármazott (D) két példányban tartalmazza a közös ős (A) adattagjait. (Egyszer a „B” ágon, egyszer a „C” ágon keresztül).
- **Kétértelműség (Ambiguity):** Ha a közös ős egy tagjára hivatkozunk a „D” objektumon keresztül, a fordító nem tudja eldönteni, melyik példányt (az „B”-ben lévőt vagy a „C”-ben lévőt) kell használni.

46.2 Strukturális Ábra (A Rombusz)



Hiba: A „D” osztályban **két darab** „id” változó keletkezik ($A_{viaB} :: id$ és $A_{viaC} :: id$).

46.3 Megoldás: Virtuális öröklődés

A problémát úgy oldjuk meg, hogy a közbenső szinteken (B és C) `virtual public` módon örököljük az őst. Ez garantálja, hogy a „D” osztályban csak **egyetlen, közös példány** jöjjön létre az „A” osztályból.

46.4 Példa a hibás és javított esetre

```
1 struct A { int adat; };
2
3 // 1. A PROBLÉMÁS ESET (Normál öröklés)
4 struct B : public A {};
5 struct C : public A {};
6 struct D : public B, public C {};
7
8 void hiba() {
9     D obj;
```

```
10      // obj.adat = 10; // HIBA! "Ambiguous" (Kétértelmű)
11
12      // Csak így érhető el (kényelmetlen és redundáns):
13      obj.B::adat = 10;
14      obj.C::adat = 20;
15  }
16
17  // ---------------------------------------------
18
19  // 2. A MEGOLDÁS (Virtuális öröklés)
20  // A 'virtual' kulcsszó biztosítja a közös példányt
21  struct V_B : virtual public A {};
22  struct V_C : virtual public A {};
23
24  // A végső osztályban (V_D) csak EGY 'A' lesz
25  struct V_D : public V_B, public V_C {};
26
27  void megoldas() {
28      V_D obj;
29      obj.adat = 10; // MŰKÖDIK! Nincs kétértelműség.
30  }
```

47 Ismertesse a tisztán virtuális metódus készítésének szintaktikáját! Hogyan nevezzük a legalább 1 tisztán virtuális metódust tartalmazó osztályt? Milyen szabályok vonatkoznak erre az osztályra?

A C++ nyelvben a tisztán virtuális (pure virtual) metódusok szolgálnak arra, hogy egy osztályban csak a függvény interfészét (szignatúráját) határozzuk meg, a megvalósítást (implementációt) pedig kötelezően a leszármazottakra bízzuk.

47.1 Szintaktika

A tisztán virtuális metódust a deklaráció végére írt `= 0` jelöléssel (pure-specifier) hozzuk létre.

- A függvénynek általában nincs törzse (implementációja) az adott osztályban.
- A `virtual` kulcsszó használata kötelező.

```
class Alakzat {
public:
    // Tisztán virtuális metódus
    // A "= 0" jelzi, hogy itt nincs implementáció
    virtual double terulet() const = 0;

    // Virtuális destruktor (ajánlott)
    virtual ~Alakzat() {}
};
```

47.2 Elnevezés

Azt az osztályt, amely legalább egy tisztán virtuális metódust tartalmaz, **absztrakt osztálynak** (Abstract Class) nevezzük. Gyakran használják őket interfészként (Interface), ahol az összes metódus tisztán virtuális.

47.3 Az absztrakt osztályra vonatkozó szabályok

Az absztrakt osztályok viselkedése eltér a hagyományos (konkrét) osztályokétól:

- **Példányosítás tilalma:** Absztrakt osztályból közvetlenül **nem hozható létre objektum** (példány).

```
Alakzat a; // HIBA: absztrakt osztály nem példányosítható
```

- **Pointerek és Referenciák:** Bár példány nem hozható létre, **mutató (pointer) vagy referencia** típusként használható. Ez teszi lehetővé a polimorfizmust.

```
Alakzat* mutato = new Kor(); // HELYES (ha a Kor konkrét)
```

- **Leszármaztatási kötelezettség:** Ha egy leszármazott osztály nem definiálja felül (override) az összes örökölt tisztán virtuális metódust (nem ad nekik törzset), akkor a leszármazott osztály is **absztrakt marad**, és nem lehet példányosítani.
- **Adattagok és konkrét metódusok:** Az absztrakt osztály tartalmazhat normál (nem tisztán virtuális) tagfüggvényeket és adattagokat is, amelyek a közös logikát valósítják meg.

48 Ismertesse az „overload” és „override” elvek közti különbséget, amennyiben ős és leszármazottban történő előfordulásról van szó!

A C++ programozásban gyakran kevert két fogalom a függvények túlterhelése (overloading) és felüldefiniálása (overriding). Bár mindkettő azonos nevű függvényekkel dolgozik, a működési mechanizmusuk és céljuk alapvetően eltérő, különösen öröklődési viszonyban.

48.1 1. Overload (Túlterhelés)

A túlterhelés azt jelenti, hogy több azonos nevű, de eltérő paraméterlistájú függvény létezik.

- **Szabály:** A név azonos, de a szignatúrának (paraméterek száma vagy típusa) különböznie kell.
- **Kötés ideje:** Fordítási időben dől el (Korai kötés / Static polymorphism).
- **Öröklődésnél (Veszélyforrás):**
  - Ha a leszármazottban létrehozunk egy függvényt ugyanazzal a névvel, de más paraméterekkel, az **elfedi (hide)** az ősosztály azonos nevű függvényeit.
  - Ez technikailag *név elfedés* (name hiding), nem klasszikus túlterhelés, hacsak nem használjuk a `using Base::fgv;` utasítást a láthatóság visszaállítására.

48.2 2. Override (Felüldefiniálás)

A felüldefiniálás azt jelenti, hogy a leszármazott osztály megváltoztatja (lecseréli) az ősosztálytól örökölt viselkedést.

- **Szabály:** A névnek és a paraméterlistának (szignatúrának) **teljesen egyeznie** kell.
- **Feltétel:** Az ősosztályban a függvénynek `virtual`-nak kell lennie.
- **Kötés ideje:** Futásidőben dől el (Késői kötés / Dynamic polymorphism).
- **Cél:** A polimorfizmus megvalósítása (ős típusú mutatón keresztül a specifikus metódus hívása).

48.3 Összehasonlító táblázat

Tulajdonság	Overload (Túlterhelés)	Override (Felüldefiniálás)
Hatókör	Általában egy osztályon belül (vagy `using`-gal)	Öröklődési láncban (Ős ↔ Utód)
Szignatúra	**Különböző** kell legyen	**Azonosnak** kell lennie
Kulcsszó	Nincs speciális kulcsszó	`virtual` (ősben), `override` (utódban)
Kötés	Statikus (Fordítási idő)	Dinamikus (Futási idő)
Visszatérés	Eltérhet (de önmagában nem elég)	Egyeznie kell (vagy kovariáns lehet)

48.4 Példa a két esetre

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

class Os {
public:
    // Virtuális függvény -> Override-olható
```

```
7       virtual void kiir() {
8           std::cout << "Os: parameter nelkul" << std::endl;
9       }
10
11      // Sima függvény -> Overload-olható
12      void szamol(int a) {
13          std::cout << "Os szamol: " << a << std::endl;
14      }
15  };
16
17  class Utod : public Os {
18  public:
19      // 1. OVERRIDE (Felüldefiniálás)
20      // Ugyanaz a név és paraméter, 'virtual' az ősben.
21      // Lecseréli az ős viselkedését polimorf használatkor.
22      void kiir() override {
23          std::cout << "Utod: parameter nelkul" << std::endl;
24      }
25
26      // 2. OVERLOAD (Túlterhelés / Elfedés)
27      // Ugyanaz a név, de MÁS paraméter.
28      // FIGYELEM: Ez elrejti az ős 'szamol(int)' függvényét!
29      void szamol(std::string s) {
30          std::cout << "Utod szoveggel: " << s << std::endl;
31      }
32
33      // Ha el akarjuk érni az int-es változatot is az Utód objektumon:
34      using Os::szamol;
35  };
36
37  int main() {
38      Utod u;
39      u.kiir();          // Utod::kiir (Override miatt)
40      u.szamol("Szia");  // Utod::szamol(string)
41
42      // u.szamol(10);   // HIBA lenne a 'using' sor nélkül (elfedés miatt)
43      u.szamol(10);      // Így már működik (Overload)
44
45      return 0;
46  }
```

49 Ismertesse a „static_cast” és „dynamic_cast” kulcsszavak működését! Hol fordulhat elő hibásan interpretált memória-terület?

A C++ nyelvben a típuskonverziók (casting) biztonságosabbá tételére vezették be az új típusú kasztoló operátorokat a C-stílusú `(type)value` helyett. A két leggyakrabban használt operátor eltérő időben és módon működik.

49.1 static_cast (Fordítási idejű konverzió)

Ez a legáltalánosabb konverzió, amely fordítási időben (compile-time) történik.

- **Működés:** A fordítóra bízza a konverzió elvégzését az ismert típusinformációk alapján. Nincs futásidejű ellenőrzés (overhead).
- **Felhasználás:**
  - Numerikus típusok között (pl. `float` → `int`).
  - Öröklődési láncban felfelé (Upcast): Mindig biztonságos.
  - Öröklődési láncban lefelé (Downcast): **Veszélyes**, mert a fordító nem ellenőrzi, hogy a mutató ténylegesen arra a típusra mutat-e.

49.2 dynamic_cast (Futásidejű konverzió)

Kifejezetten polimorf osztályok (ahol van legalább egy `virtual` függvény) közötti biztonságos navigációra szolgál.

- **Működés:** Futásidőben (runtime) ellenőrzi az objektum valódi típusát az RTTI (Run-Time Type Information) segítségével.
- **Feltétel:** Az osztálynak polimorfnak kell lennie (kell `vtable`).
- **Eredmény:**
  - **Pointer esetén:** Ha a konverzió sikertelen (az objektum nem a kért típusú), `nullptr`-t ad vissza.
  - **Referencia esetén:** Ha sikertelen, `std::bad_cast` kivételt dob.
- **Költség:** Lassabb a `static_cast`-nál a futásidejű ellenőrzés miatt.

49.3 Hibásan interpretált memória-terület (Veszélyforrás)

A hibás memória-interpretáció a **`static_cast` helytelen használatakor**, kifejezetten a „Downcast” (ős → utód konverzió) során fordulhat elő.

- **A szituáció:** Van egy `Base*` mutatónk, amely valójában egy `Base` példányra (vagy egy másik ágon lévő `DerivedA`-ra) mutat.
- **A hiba:** Ezt a mutatót `static_cast`-tal kényszerítjük egy `DerivedB*` típusra.
- **Következmény (Undefined Behavior):**
  - A fordító elfogadja a kérést („te tudod, mit csinálsz” alapon).
  - Amikor a program megpróbálja elérni a `DerivedB` egyedi adattagjait a mutatón keresztül, a memóriában lévő **szemetet** vagy **más változók bájtjait** fogja az adott adattagként értelmezni.
  - Ez memóriasérüléshez vagy összeomláshoz vezet.

49.4 Példa a működésre és a hibára

```cpp
#include <iostream>

class Base { public: virtual ~Base() {} }; // Polimorf ős
class DerivedA : public Base { public: int a_data = 10; };
class DerivedB : public Base { public: int b_data = 20; };

int main() {
    Base* ptr = new DerivedA(); // Tényleges típus: A

    // 1. dynamic_cast (Biztonságos)
    // Megpróbáljuk B-ként kezelni -> SIKERTELEN lesz
    DerivedB* b_safe = dynamic_cast<DerivedB*>(ptr);
    if (b_safe == nullptr) {
        std::cout << "Dynamic cast: Ez nem DerivedB!" << std::endl;
    }

    // 2. static_cast (Veszélyes / Hibás)
    // Kényszerítjük, hogy B-ként kezelje az A-t.
    // A fordító engedi, de a futáskor baj lesz.
    DerivedB* b_unsafe = static_cast<DerivedB*>(ptr);

    // HIBA: Olyan memóriaterületet olvasunk 'b_data' néven,
    // ami valójában az 'DerivedA' objektum része (vagy szemét).
    // Ez a "hibásan interpretált memória".
    std::cout << "Static cast ertek (szemet): " << b_unsafe->b_data << std::endl;

    delete ptr;
    return 0;
}
```

50 Ismertesse egy előre megírt programrendszer (netről letöltött, vagy eszközzel kapott SDK) használatának lépéseit C++-ban!

Egy külső könyvtár (Library) vagy szoftverfejlesztői készlet (SDK) integrálása C++-ban több lépésből áll, mivel a nyelv szétválasztja a deklarációt (fejlécfájlok) és a megvalósítást (lefordított binárisok). A folyamat három fő fázisra bontható: fordítási beállítások, szerkesztési (linkelési) beállítások és futtatási környezet.

50.1 1. Az állományok előkészítése

A letöltött SDK általában három fő mappát tartalmaz, amelyeket érdemes a projekt közelében elhelyezni:

- **include:** A `.h` vagy `.hpp` kiterjesztésű fejlécfájlok (interfész leírása).
- **lib:** A lefordított könyvtárfájlok (`.lib` Windows-on, `.a` Linux-on).
- **bin:** A futtatható binárisok vagy dinamikus könyvtárak (`.dll` Windows-on, `.so` Linux-on).

50.2 2. Fordítási beállítások (Compiler Settings)

A fordítónak tudnia kell, hol találja a függvények és osztályok deklarációit, hogy értelmezni tudja a kódban lévő hívásokat.

- **Include Directories (Keresési útvonal):** Meg kell adni a fordítónak (IDE beállításokban vagy `-I` flaggel) az SDK **include** mappájának elérési útját.
- **Forráskód:** A saját kódban be kell emelni a szükséges fejlécfájlt:

```
1 #include <sdk_header.h> // Vagy "sdk_header.h"
```

50.3 3. Szerkesztési beállítások (Linker Settings)

A szerkesztőnek (linker) össze kell kötnie a lefordított kódunkat az SDK lefordított kódjával. Ha ez a lépés kimarad, "Unresolved external symbol" hibát kapunk.

- **Library Directories (Könyvtár útvonal):** Meg kell adni a linkernek (IDE beállításokban vagy `-L` flaggel) az SDK **lib** mappájának elérési útját.
- **Input Dependencies (Függőségek):** Konkrétan meg kell nevezni, melyik `.lib` fájlt kell hozzászerkeszteni a programhoz (pl. `mylib.lib` vagy `-lmylib`).

50.4 4. Futásidejű feltételek (Runtime)

Ez a lépés attól függ, hogy statikus vagy dinamikus linkelést használunk-e.

- **Statikus linkelés (.lib / .a):** A kód belemásolódik a mi `.exe` fájlunkba. Nincs további teendő, de a program mérete nagyobb lesz.
- **Dinamikus linkelés (.dll / .so):** Csak hivatkozások kerülnek a programba.
  - A program indításakor az operációs rendszernek meg kell találnia a dinamikus könyvtárat.
  - **Megoldás:** A szükséges `.dll` fájlokat (a **bin** mappából) be kell másolni a kész programunk (`.exe`) mellé, vagy hozzáadni az elérési utat a rendszer `PATH` környezeti változójához.

50.5 Összefoglaló példa (CMake stílusban)

```
# 1. Include mappa megadása (Fordító)
include_directories(path/to/sdk/include)

# 2. Lib mappa megadása (Linker)
link_directories(path/to/sdk/lib)

# 3. Futtatható fájl létrehozása
add_executable(MyGame main.cpp)

# 4. Konkrét lib fájl hozzárendelése (Linker)
target_link_libraries(MyGame sdk_library_name)
```

51 Ismertesse a vector STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)!

Az `std::vector` a C++ Standard Template Library (STL) legfontosabb és leggyakrabban használt soros tárolója. Lényegében egy dinamikusan méreteződő tömböt valósít meg, amely ötvözi a statikus tömbök gyors elérését a listák rugalmasságával.

51.1 Memória modell

A `vector` legfontosabb tulajdonsága a fizikai memóriában való elhelyezkedése:

- **Folytonos tárterület (Contiguous memory):** Az elemeket a memóriában egymás után, megszakítás nélkül tárolja (mint egy C tömb).
- **Előnyei:**
  - **Gyors elérés:** Támogatja a pointer-aritmetikát.
  - **Cache-barát:** A processzor gyorsítótára hatékonyan tudja előtölteni az adatokat (spatial locality).
  - **Kompatibilitás:** Átadhatjuk olyan C függvényeknek, amelyek nyers tömböt (`T*`) várnak (ehhez a `data()` tagfüggvényt használjuk).
- **Méretek:**
  - Egy vektor objektum a stack-en csak egy „adminisztrációs” részt tárol (általában 3 pointert: kezdet, vége, foglalt terület vége), a tényleges adatok a **heap**-en (kupacon) vannak.

51.2 Bővíthetőség és Kapacitás

A vektor automatikusan kezeli a memóriafoglalást, de fontos megkülönböztetni két fogalmat:

- **Size (Méret):** A ténylegesen tárolt elemek száma (`size()`).
- **Capacity (Kapacitás):** A lefoglalt, de még nem feltétlenül használt memória mérete (`capacity()`).

Az újrafoglalás (Reallocation) mechanizmusa: Amikor új elemet adunk hozzá (`push_back`), és `size == capacity`:

1. A vektor egy **új**, nagyobb memóriaterületet foglal le (általában a korábbi méret 2-szeresét).
2. **Átmásolja** (vagy mozgatja) a régi elemeket az új helyre.
3. **Felszabadítja** a régi memóriaterületet.
4. Beilleszti az új elemet.

Megjegyzés: Ez a művelet költséges ($O(N)$), de mivel ritkán történik, a beszúrás átlagos (amortizált) költsége konstans ($O(1)$). A felesleges másolások elkerülésére használjuk a `reserve(n)` függvényt, ha előre sejtjük az elemszámot.

51.3 Bejárás és Elemek elérése

Mivel a memória folytonos, a vektor támogatja a **véletlen elérést (Random Access)**.

- **Indexelés (`[]`):** Gyors ($O(1)$), de nem végez határellenőrzést (túlindexelés esetén undefined behavior).
- **Biztonságos elérés (`at()`):** Határellenőrzést végez, hiba esetén `std::out_of_range` kivételt dob (lassabb).
- **Iterátorok:** *Random Access Iterator*-t biztosít (`begin()`, `end()`), így pointer-szerűen léptethető és tetszőleges távolságra ugorhatunk vele.

51.4 Példa a működésre

```cpp
#include <iostream>
#include <vector>

int main() {
    // 1. Létrehozás
    std::vector<int> szamok;

    // Optimalizáció: Előre foglalunk helyet 10 elemnek
    // Így elkerüljük a többszöri átmásolást a ciklusban
    szamok.reserve(10);

    // 2. Feltöltés (Bővítés)
    for (int i = 0; i < 5; ++i) {
        szamok.push_back(i * 10);
        // Itt a size nő, a capacity 10 marad
    }

    // 3. Bejárás (Range-based for loop - C++11)
    std::cout << "Elemek: ";
    for (const int& x : szamok) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    // 4. Memória állapot
    std::cout << "Size: " << szamok.size() << std::endl;         // 5
    std::cout << "Capacity: " << szamok.capacity() << std::endl; // 10

    // 5. Közvetlen elérés
    szamok[2] = 99; // 3. elem módosítása

    return 0;
}
```

52 Ismertesse a deque STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)!

Az `std::deque` (Double Ended Queue – kétvégű sor) az STL egyik speciális szekvenciális tárolója. Fő jellemzője, hogy a vektorral ellentétben nemcsak a végén, hanem az elején is hatékonyan bővíthető.

52.1 Memória modell (Szegmentált felépítés)

A `deque` legfontosabb tulajdonsága, hogy a memóriában **nem folytonosan** helyezkedik el (ellentétben a `vector`-ral).

- **Blokkos szerkezet:** Az elemeket több, kisebb, fix méretű memóriablokkban (chunk/buffer) tárolja.
- **Központi vezérlő (Map):** Egy belső "térképet" (pointerek tömbjét) tart fenn, amely ezekre a blokkokra mutat.
- **Nincs pointer-kompatibilitás:** Mivel a memória nem folytonos, a `deque` elemei nem adhatók át közvetlenül olyan C-függvényeknek, amelyek nyers tömböt várnak (nincs `.data()` függvény, ami az egészet visszaadná).

52.2 Bővíthetőség

A `deque` a dinamikus memóriakezelést a vektorhoz képest eltérő stratégiával valósítja meg:

- **Kétirányú bővítés:** Támogatja a `push_back()` és a `push_front()` műveleteket is, mindkettő átlagos műveletigénye $O(1)$.
- **Nincs teljes másolás:** Ha betelik a tárterület, nem kell az összes meglévő elemet új helyre másolni (mint a vektornál). Csak egy új memóriablokkot kell lefoglalni és hozzárendelni a központi térképhez.
- **Referencia érvényesség:** Bővítéskor az elemek a memóriában a helyükön maradnak, így a rájuk mutató referenciák és pointerek érvényesek maradnak (bár az iterátorok érvénytelenné válhatnak, ha a központi map-et át kell szervezni).

52.3 Bejárás és Elérés

A bonyolultabb belső szerkezet ellenére a `deque` kifelé hasonlóan viselkedik, mint a vektor.

- **Véletlen elérés (Random Access):** Támogatja az indexelést (`operator[]` és `at()`) $O(1)$ időben.
  - *Megjegyzés:* Ez valamivel lassabb, mint a vektor indexelése, mivel két lépcsőben történik (először a megfelelő blokk kikeresése, majd azon belül az elem elérése).
- **Iterátorok:** Random Access Iteratort biztosít, tehát lehet vele ugrani (`it + 5`), de az iterátor implementációja bonyolultabb ("smart pointer"), mert tudnia kell ugrani a memóriablokkok határain.

52.4 Példa a használatra

```cpp
#include <iostream>
#include <deque>

int main() {
    std::deque<int> d;

    // 1. Bővítés a végén (mint a vector)
    d.push_back(10);
```

```cpp
    d.push_back(20);

    // 2. Bővítés az elején (ez a vectornál O(N) lenne, itt O(1))
    d.push_front(5);
    d.push_front(1);

    // Memória logika: [1, 5] (egy blokkban) ... [10, 20] (másik blokkban)
    // De számunkra folytonosnak látszik: 1, 5, 10, 20

    // 3. Véletlen elérés
    std::cout << "3. elem: " << d[2] << std::endl; // Kiírja: 10

    // 4. Bejárás
    for (int n : d) {
        std::cout << n << " ";
    }

    return 0;
}
```

53 Ismertesse a list STL tároló tulajdonságait (memória modell, bejárás, bővíthetőség)!

Az `std::list` a C++ STL kétirányú láncolt listáját (Doubly Linked List) valósítja meg. Használata akkor javasolt, ha gyakori a beszúrás és törlés a tároló közepén, de ritka a véletlenszerű elérés.

53.1 Memória modell (Láncolt szerkezet)

A `list` memóriaképe alapvetően eltér a `vector`-étól és a `deque`-étól.

- **Csomópontok (Nodes):** Minden elem egy önállóan, dinamikusan lefoglalt memóriablokkban (csomópontban) helyezkedik el.
- **Szórt elhelyezkedés:** Az elemek nem egymás mellett vannak a memóriában, hanem "szétszórva" a heap-en.
- **Belső szerkezet:** Minden csomópont három dolgot tárol:
  1. Magát az adatot.
  2. Egy mutatót az előző elemre (*prev*).
  3. Egy mutatót a következő elemre (*next*).
- **Memória overhead:** Jelentős többletmemóriát igényel elemenként a két pointer miatt (pl. egy `char` tárolása esetén a pointerek mérete sokszorosa az adatának).

53.2 Bővíthetőség és Módosítás

Ez a `list` legnagyobb erőssége.

- **Konstans idejű beszúrás/törlés ($O(1)$):** Bárhol (elején, végén, közepén) hozunk létre vagy törlünk elemet, az műveletigénye konstans (csak a pointereket kell átkötni).
- **Nincs másolás:** Beszúráskor nem kell a többi elemet elmozgatni a memóriában.
- **Iterátorok érvényessége:** Ez kritikus előny! Elem hozzáadása vagy törlése **soha nem érvényteleníti** a többi elemre mutató iterátorokat vagy pointereket (kivéve persze azt az egyet, amit épp törlünk).

53.3 Bejárás és Elérés

A láncolt szerkezet hátránya a hozzáférés módjában jelentkezik.

- **Nincs véletlen elérés (No Random Access):** Nem használható az `operator[]` vagy az `at()`. Nem lehet azt mondani, hogy "add ide az 5. elemet".
- **Szekvenciális bejárás:** Egy elem eléréséhez végig kell lépkedni az összes előtte lévőn ($O(N)$).
- **Kétirányú iterátor:** Csak `++` és `--` műveleteket támogat, aritmetikát (pl. `it + 5`) nem.
- **Cache-barátságtalan:** Mivel az elemek szétszórva vannak a memóriában, a processzor cache-t nem tudja hatékonyan kihasználni (sok "cache miss"), így a bejárás lassabb lehet, mint egy vektornál.

53.4 Példa a használatra

```cpp
#include <iostream>
#include <list>
#include <algorithm>

int main() {
    std::list<int> l;

    // 1. Feltöltés (O(1))
    l.push_back(10);
    l.push_front(5);
    l.push_back(20); // Lista: 5 <-> 10 <-> 20

    // 2. Beszúrás középre
    // Megkeressük a 10-es elemet (ez lassú, O(N))
    auto it = std::find(l.begin(), l.end(), 10);

    // Beszúrjuk elé a 7-est (ez gyors, O(1))
    if (it != l.end()) {
        l.insert(it, 7);
    }
    // Lista: 5 <-> 7 <-> 10 <-> 20

    // 3. Bejárás
    // Nincs l[2], csak iterátoros bejárás
    std::cout << "Lista elemei: ";
    for (int x : l) {
        std::cout << x << " ";
    }
    std::cout << std::endl;

    // 4. splice: Lista darabok átmozgatása (speciális lista művelet)
    // Egy másik lista tartalmát átfűzi ide másolás nélkül O(1)
    std::list<int> l2 = {100, 200};
    l.splice(l.begin(), l2);

    return 0;
}
```

54 Ismertesse a set, multiset STL tárolók tulajdonságait (memória modell, bejárás, bővíthetőség)!

Az `std::set` és `std::multiset` az STL asszociatív tárolói. Fő jellemzőjük, hogy az elemeket nem a beillesztés sorrendjében, hanem az értékük szerinti **rendezett állapotban** tárolják. A `set` minden elemből csak egyet tárolhat (egyedi kulcsok), míg a `multiset` engedélyezi a duplikációkat.

54.1 Memória modell (Kiegyensúlyozott fa)

A háttérben nem tömb és nem láncolt lista áll, hanem egy bonyolultabb, csomópont-alapú adatszerkezet.

- **Adatszerkezet:** Szinte minden implementációban (GCC, MSVC) **Piros-Fekete fát (Red-Black Tree)** használnak. Ez egy önkiegyensúlyozó bináris keresőfa.
- **Csomópontok felépítése:** Minden elem külön memóriablokkban helyezkedik el (mint a listánál), amely tartalmazza:
  - Magát az adatot (értéket).
  - Három mutatót: Szülő (Parent), Bal gyerek (Left), Jobb gyerek (Right).
  - Egy színt (Piros vagy Fekete) a fa egyensúlyának fenntartásához.
- **Memória overhead:** Jelentős, mivel minden adathoz több pointert és adminisztrációs bitet kell tárolni.

54.2 Bővíthetőség és Teljesítmény

Mivel az adatszerkezet mindig rendezett marad, a beszúrás és törlés költségesebb, mint egy listánál, de a keresés sokkal gyorsabb.

- **Logaritmikus műveletigény ($O(\log N)$):**
  - Beszúrás (`insert`): Megkeresi a helyét a fában, majd szükség esetén forgatásokkal kiegyensúlyozza azt.
  - Törlés (`erase`): Szintén logaritmikus.
  - Keresés (`find`): Bináris keresés elvén működik, nagyon gyors.
- **Iterátor érvényesség:** Elem beszúrása vagy törlése **nem érvényteleníti** a többi elemre mutató iterátorokat (mivel a csomópontok címe nem változik a memóriában, csak a pointerek kötődnek át).
- **Módosíthatatlanság:** A tárolt elemek kulcsai **konstansok**. Nem módosíthatunk egy elemet közvetlenül a tárolóban (mivel az elrontaná a sorrendet). Módosításhoz törölni kell a régit, és beszúrni az újat.

54.3 Bejárás és Elérés

- **Rendezett bejárás:** Az iterátorok (`begin()` → `end()`) "In-order" bejárást végeznek, így az elemeket mindig növekvő sorrendben kapjuk vissza.
- **Típus:** Kétirányú iterátor (Bidirectional Iterator). Csak `++` és `--` művelet van, ugrani nem lehet.
- **Nincs indexelés:** Nem használható a `[]` operátor vagy az `at()`, mivel nincs véletlen elérés.

54.4 Példa a működésre

```cpp
#include <iostream>
#include <set>

int main() {
    // 1. Set létrehozása (automatikusan rendezett, egyedi)
    std::set<int> halmaz;

    // 2. Beszúrás O(log N)
    halmaz.insert(40);
    halmaz.insert(10);
    halmaz.insert(40); // DUPLIKÁCIÓ: A set ezt figyelmen kívül hagyja!
    halmaz.insert(20);

    // 3. Bejárás (Mindig rendezett sorrendben!)
    std::cout << "Set elemei: ";
    for (int x : halmaz) {
        std::cout << x << " ";
    }
    // Kimenet: 10 20 40 (Növekvő sorrend, duplikáció nélkül)
    std::cout << std::endl;

    // 4. Multiset példa
    std::multiset<int> multi;
    multi.insert(40);
    multi.insert(40); // Ez bekerül másodszor is

    std::cout << "Multiset 40 darabszama: " << multi.count(40) << std::endl; // 2

    // 5. Keresés O(log N)
    auto it = halmaz.find(20);
    if (it != halmaz.end()) {
        std::cout << "Megtalalva: " << *it << std::endl;
        // *it = 25; // HIBA! Az elemek read-only (const) típusúak
    }

    return 0;
}
```

55 Ismertesse a map, multimap STL tárolók tulajdonságait (memória modell, bejárás, bővíthetőség)!

Az `std::map` és `std::multimap` a C++ STL asszociatív tárolói, amelyek **kulcs-érték párokat** (Key-Value pairs) tárolnak.

- **std::map:** Minden kulcs egyedi (egy kulcshoz egy érték tartozik). Gyakran hívják asszociatív tömbnek vagy szótárnak.
- **std::multimap:** Egy kulcshoz több érték is tartozhat (duplikált kulcsok engedélyezettek).

55.1 Memória modell (Fa szerkezet)

Hasonlóan a `set`-hez, a háttérben itt is egy kiegyensúlyozott bináris keresőfa, jellemzően **Piros-Fekete fa (Red-Black Tree)** áll.

- **Elemek szerkezete:** Minden csomópont egy `std::pair<const Key, T>` objektumot tárol.
  - **Key (Kulcs):** Konstans, nem módosítható (mivel ez határozza meg a helyét a fában).
  - **Value (Érték):** Módosítható adattag.
- **Elhelyezkedés:** A csomópontok a memóriában szétszórva, dinamikusan foglalódnak le, pointerek kötik össze őket.
- **Rendezés:** A tároló az elemeket a **kulcsok** szerint automatikusan rendezve tartja.

55.2 Bővíthetőség és Műveletek

A műveletek sebességét a fa magassága határozza meg.

- **Logaritmikus idő ($O(\log N)$):** Beszúrás, törlés és keresés (kulcs alapján).
- **Indexelő operátor (`[]`) - Csak `map`:**
  - `map[kulcs] = ertek;` formában használható.
  - **Fontos mellékhatás:** Ha a kulcs még nem létezik, **létrehozza** azt az alapértelmezett értékkel, majd visszaadja a referenciáját. Ha csak keresni akarunk, használjuk a `find()`-ot vagy `at()`-et!
  - A `multimap` nem támogatja a `[]` operátort, mivel egy kulcshoz több érték is tartozhatna.
- **Iterátor érvényesség:** Stabil. Elem beszúrása vagy törlése nem érvényteleníti a többi elemre mutató iterátorokat.

55.3 Bejárás

- **Rendezett sorrend:** Az iterátorok a kulcsok növekvő sorrendjében járják be a párokat.
- **Hozzáférés:** Az iterátor egy `pair`-re mutat:
  - `it->first`: A kulcs (const).
  - `it->second`: Az érték (módosítható).
- **Multimap tartomány:** A `multimap`-ben az azonos kulcsú elemek egymás után helyezkednek el. Ezeket az `equal_range()` függvénnyel lehet egyben lekérdezni.

55.4 Példa a használatra

```cpp
#include <iostream>
#include <map>
#include <string>

int main() {
    // 1. Map létrehozása (Kulcs: string, Érték: int)
    std::map<std::string, int> eletkorok;

    // 2. Beszúrás indexelő operátorral (csak map)
    eletkorok["Anna"] = 25; // Létrehozás
    eletkorok["Bela"] = 30;
    eletkorok["Anna"] = 26; // Módosítás (felülírás)

    // Beszúrás insert metódussal (párként)
    eletkorok.insert(std::make_pair("Cecil", 40));

    // 3. Keresés
    // A [] operátor létrehozná, ha nincs, ezért find-ot használunk ellenőrzésre
    auto it = eletkorok.find("Bela");
    if (it != eletkorok.end()) {
        std::cout << "Bela kora: " << it->second << std::endl;
        it->second = 31; // Érték módosítható
        // it->first = "Geza"; // HIBA! Kulcs nem módosítható
    }

    // 4. Bejárás (Rendezett: Anna -> Bela -> Cecil)
    std::cout << "Névsor:" << std::endl;
    for (const auto& par : eletkorok) {
        std::cout << par.first << ": " << par.second << std::endl;
    }

    // 5. Multimap példa
    std::multimap<std::string, int> osztalyzatok;
    osztalyzatok.insert({"Matek", 5});
    osztalyzatok.insert({"Matek", 4}); // Engedi a duplikált kulcsot

    std::cout << "Matek jegyek szama: " << osztalyzatok.count("Matek") << std::endl; // 2

    return 0;
}
```

56 Ismertesse az STL tároló adaptereket és működésüket! Melyik mire használható?

Az STL tároló adapterek (Container Adapters) nem önálló adatszerkezetek, hanem meglévő szekvenciális tárolókra (pl. `vector`, `deque`, `list`) épülő burkoló osztályok. Céljuk, hogy korlátozzák és specializálják a hozzáférést egy bizonyos működési elv (pl. LIFO vagy FIFO) szerint.

Közös jellemzőjük, hogy **nem támogatják az iterátorokat**, tehát az elemeket nem lehet szabadon bejárni, csak a szigorú szabályok szerint elérni.

56.1 1. std::stack (Verem)

A **LIFO** (Last In, First Out – Utolsóként be, elsőként ki) elvet valósítja meg.

- **Működés:** Elemeket csak a "tetejére" lehet helyezni, és csak onnan lehet elvenni.
- **Alapértelmezett tároló:** `std::deque` (de használható `vector` vagy `list` is).
- **Fő metódusok:**
  - `push()`: Elem felhelyezése.
  - `pop()`: A legfelső elem eltávolítása (visszatérési érték nélkül).
  - `top()`: A legfelső elem lekérdezése (referenciát ad).
- **Felhasználás:**
  - Visszavonás (Undo) műveletek.
  - Függvényhívási lánc (Call stack) szimulálása.
  - Szintaktikai elemzés (pl. zárójelek párba állítása).

56.2 2. std::queue (Sor)

A **FIFO** (First In, First Out – Elsőként be, elsőként ki) elvet valósítja meg.

- **Működés:** Az egyik végén (back) helyezzük be az elemeket, a másik végén (front) vesszük ki.
- **Alapértelmezett tároló:** `std::deque` (lehet `list` is, de `vector` NEM, mert az elején való törlés nem hatékony).
- **Fő metódusok:**
  - `push()`: Elem hozzáadása a sor végére.
  - `pop()`: Elem eltávolítása a sor elejéről.
  - `front()`: Az első elem lekérdezése.
  - `back()`: Az utolsó elem lekérdezése.
- **Felhasználás:**
  - Pufferelés (Producer-Consumer probléma).
  - Nyomtatási feladatok ütemezése.
  - Szélességi bejárás (BFS) gráfokban.

56.3 3. std::priority_queue (Prioritásos sor)

[Image of binary heap data structure]

Olyan sor, ahol az elemek nem érkezési, hanem **prioritási sorrendben** (alapértelmezésben csökkenő, azaz a legnagyobb van elöl) jönnek ki.

- **Működés:** A háttérben egy **Heap (Kupac)** adatszerkezetet tart fenn. A beszúrás és kivétel logaritmikus idejű ($O(\log N)$).
- **Alapértelmezett tároló:** `std::vector`.
- **Fő metódusok:**
  - `push()`: Elem beszúrása (a helyére kerül a prioritás szerint).
  - `pop()`: A legmagasabb prioritású elem eltávolítása.
  - `top()`: A legmagasabb prioritású elem lekérdezése.
- **Felhasználás:**
  - Dijkstra legrövidebb út algoritmus.
  - Operációs rendszer feladatütemezője.

56.4 Példa a használatra

```cpp
#include <iostream>
#include <stack>
#include <queue>
#include <vector>

int main() {
    // 1. Stack (LIFO)
    std::stack<int> s;
    s.push(10);
    s.push(20);
    // Teteje: 20, Alatta: 10
    std::cout << "Stack teteje: " << s.top() << std::endl; // 20
    s.pop(); // 20 torlese

    // 2. Queue (FIFO)
    std::queue<int> q;
    q.push(10);
    q.push(20);
    // Eleje: 10, Vege: 20
    std::cout << "Sor eleje: " << q.front() << std::endl; // 10
    q.pop(); // 10 torlese

    // 3. Priority Queue (Max Heap)
    std::priority_queue<int> pq;
    pq.push(10);
    pq.push(50);
    pq.push(30);
    // A legnagyobb kerul legelore automatikusan
    std::cout << "Legnagyobb elem: " << pq.top() << std::endl; // 50
```

```
30
31      return 0;
32  }
```

57 Ismertesse az STL iterátorok működését és feladatát egy lista STL tároló esetén!

Az iterátorok az STL (Standard Template Library) „ragasztóanyagai”, amelyek egységes felületet biztosítanak a különböző tárolók (konténerek) bejárására anélkül, hogy a programozónak ismernie kellene a tároló belső memóriaszerkezetét.

57.1 Az iterátor feladata és fogalma

- **Absztrakció:** Az iterátor egy általánosított mutató (smart pointer). Úgy viselkedik, mint egy pointer, de a háttérben elrejti a bonyolult léptetési logikát.
- **Egységes interfész:** Lehetővé teszi, hogy ugyanazt az algoritmust (pl. `std::find`, `std::sort`) használhassuk vektoron, listán vagy set-en, mivel az algoritmus nem a tárolót, hanem annak iterátorait használja.
- **Kapcsolat:** A `begin()` (első elemre mutató) és `end()` (utolsó utáni elemre mutató) tagfüggvényekkel kérhetők el.

57.2 Működés std::list esetén (Láncolt lista)

Mivel az `std::list` elemei a memóriában szétszórtan helyezkednek el (csomópontokban), az iterátor működése eltér a vektorétól:

- **Belső szerkezet:** A lista iterátora a háttérben egy mutatót tárol az aktuális *csomópontra* (node).
- **Léptetés (`++it`):** Amikor az iterátort növeljük, az nem a memóriacímet növeli (mint egy tömbnél), hanem a csomópont `next` pointerét követi:
  `current = current->next;`
- **Dereferálás (`*it`):** A `*` operátor az aktuális csomópontban tárolt *adatot* adja vissza.

57.3 Az iterátor kategóriája: Kétirányú (Bidirectional)

A láncolt lista szerkezete miatt a lista iterátora korlátozottabb, mint a vektoré:

- **Amit TUD:**
  - Előre lépni (`++it`).
  - Hátra lépni (`-it`).
  - Egyenlőséget vizsgálni (`it1 == it2`).
- **Amit NEM TUD (No Random Access):**
  - Nem lehet tetszőlegeset ugrani (`it + 5` **tilos**).
  - Nem lehet indexelni (`it[3]` **tilos**).
  - Nem lehet a távolságot egyszerű kivonással megkapni (`it2 - it1` **tilos**).

57.4 Iterátor érvényesség (Iterator Validity)

A lista egyik legnagyobb előnye az iterátorok stabilitása:

- **Beszúráskor:** Egyik létező iterátor sem válik érvénytelenné (mivel a meglévő csomópontok nem mozdulnak el a memóriában).
- **Törléskor:** Csak az az egy iterátor válik érvénytelenné, amelyik a törölt elemre mutatott. A többi továbbra is használható.

57.5 Példa a használatra

```cpp
#include <iostream>
#include <list>
#include <algorithm> // std::advance miatt

int main() {
    std::list<int> l = {10, 20, 30, 40};

    // 1. Iterátor lekérése
    std::list<int>::iterator it = l.begin();

    // 2. Bejárás és módosítás
    while (it != l.end()) {
        *it += 1; // Dereferálás: érték módosítása
        std::cout << *it << " ";
        it++;     // Léptetés (next pointer követése)
    }

    // 3. Ugrás (Mivel nincs Random Access, ez trükkös)
    it = l.begin();
    // it = it + 2; // HIBA! Ez listánál nem működik!

    // Helyette lépésenként kell haladni (vagy segédfüggvénnyel):
    std::advance(it, 2); // O(N) idejű léptetés a háttérben

    std::cout << "\nHarmadik elem: " << *it << std::endl; // 31

    return 0;
}
```

58 Ismertesse az STL tárolókon végrehajtható algoritmusok működését és testre szabási lehetőségeiket!

Az STL (Standard Template Library) algoritmusai olyan generikus (típusfüggetlen) függvények, amelyek a tárolók elemein végeznek műveleteket (keresés, rendezés, számlálás, módosítás). A definíciók a `<algorithm>` és `<numeric>` header fájlokban találhatók.

58.1 Működési elv: Az iterátorok szerepe

Az algoritmusok legfontosabb tulajdonsága, hogy **nem ismerik a tárolót**, amin dolgoznak.

- **Elválasztás:** Az algoritmusok és a tárolók egymástól függetlenek. A kapcsolatot az **iterátorok** teremtik meg.
- **Tartomány (Range):** Az algoritmusok mindig egy `[begin, end)` intervallumon dolgoznak. A kezdő iterátor benne van a tartományban, a végzáró iterátor az utolsó elem *után* mutat.
- **Generikusság:** Mivel template-ek, bármilyen objektummal működnek, aminek van megfelelő iterátora (akár hagyományos C-tömbökkel is).
- **Korlátok:** Mivel nem érik el magát a tároló objektumot (csak az elemeit), az algoritmusok **nem tudják megváltoztatni a tároló méretét** (nem tudnak törölni vagy beszúrni, csak felülírni/cserélni), kivéve speciális *inserter* iterátorok használatával.

58.2 Algoritmus típusok

1. **Nem módosító (Non-modifying):** Pl. `find`, `count`, `for_each`. Csak olvassák az elemeket.
2. **Módosító (Modifying):** Pl. `copy`, `replace`, `transform`. Megváltoztatják az elemek értékét vagy sorrendjét.
3. **Rendezés és Halmazműveletek:** Pl. `sort`, `binary_search`, `merge`.
4. **Numerikus:** Pl. `accumulate` (összegzés).

58.3 Testre szabás (Customization)

Az algoritmusok viselkedése paraméterezhető saját logikával. Ezt úgy érjük el, hogy az adatok mellé átadunk egy függvényt vagy objektumot is.

- **Predikátumok (Predicates):** Olyan függvények, amelyek `bool` (igaz/hamis) értékkel térnek vissza. Az algoritmus ezt használja döntéshozatalra (pl. "ez az elem megfelel-e a feltételnek?").
  - **Unáris:** Egy paramétert vár (pl. `find_if`).
  - **Bináris:** Két paramétert vár (pl. rendezésnél összehasonlítás).
- **Megvalósítási formák:**
  1. **Függvény pointer:** Hagyományos C-stílusú függvény címe.
  2. **Funktor (Function Object):** Olyan osztály, amely túlterheli az `operator()`-t. Állapotot is tud tárolni.
  3. **Lambda kifejezés (C++11):** Helyben definiált, névtelen függvény. Ez a modern és leggyakoribb megoldás.

58.4 Példa: Rendezés és Keresés testre szabása

```cpp
#include <iostream>
#include <vector>
#include <algorithm> // Algoritmusok
#include <numeric>   // accumulate

int main() {
    std::vector<int> v = {10, 5, 8, 20, 3};

    // 1. Alapértelmezett működés: Növekvő rendezés
    // Csak a tartományt adjuk át
    std::sort(v.begin(), v.end());
    // v: 3, 5, 8, 10, 20

    // 2. Testre szabás Lambda kifejezéssel: Csökkenő rendezés
    // A 3. paraméter egy bináris predikátum (komparátor)
    std::sort(v.begin(), v.end(), [](int a, int b) {
        return a > b; // Ha 'a' nagyobb, kerüljön előre
    });
    // v: 20, 10, 8, 5, 3

    // 3. Feltételes keresés (find_if)
    // Keressük meg az első 6-nál nagyobb páros számot
    auto it = std::find_if(v.begin(), v.end(), [](int n) {
        return (n > 6) && (n % 2 == 0);
    });

    if (it != v.end()) {
        std::cout << "Talalat: " << *it << std::endl; // 20 vagy 10 vagy 8 (sorrendtől függ)
    }

    // 4. Művelet minden elemen (for_each)
    // Referenciát kap, így módosítani is tud
    std::for_each(v.begin(), v.end(), [](int& n) {
        n *= 2; // Minden elemet megduplázunk
    });

    return 0;
}
```

59 Ismertesse a „nyelvi változó”, „karakterisztikus függvény” és a „tagsági függvény” fogalmakat Zadeh szerint!

Lotfi A. Zadeh 1965-ben publikálta a Fuzzy (elmosódott) halmazok elméletét, amely a kétértékű (bináris) logika kiterjesztése. A tétel a hagyományos halmazelmélet és a fuzzy logika közti matematikai különbségeket, valamint az emberi gondolkodás modellezésének alapjait tárgyalja.

59.1 1. Karakterisztikus függvény (Hagyományos halmazok)

A klasszikus (úgynevezett „crisp” vagy éles) halmazelméletben egy elem vagy beletartozik egy halmazba, vagy nem. Nincs átmenet.

- **Definíció:** Legyen X az alaphalmaz (univerzum) és $A \subseteq X$ egy részhalmaz. Az A halmaz χ_A karakterisztikus függvénye:
$$\chi_A(x) = \begin{cases} 1, & \text{ha } x \in A \\ 0, & \text{ha } x \notin A \end{cases}$$
- **Jellemzői:**
  - **Értékkészlet:** $\{0, 1\}$.
  - **Jelentése:** Egyértelmű hovatartozás (igaz/hamis).
  - **Korlátja:** Nem képes modellezni a bizonytalan fogalmakat (pl. „magas ember”, „meleg idő”), ahol a határvonal nem éles.

59.2 2. Tagsági függvény (Fuzzy halmazok)

Zadeh felismerése, hogy a valóságban a tulajdonságok fokozatosak. A tagsági függvény (Membership Function) a karakterisztikus függvény általánosítása.

- **Definíció:** Legyen X az univerzum. Egy A fuzzy halmazt a μ_A tagsági függvény definiál:
$$\mu_A(x) : X \to [0, 1]$$
- **Jellemzői:**
  - **Értékkészlet:** A $[0, 1]$ zárt intervallum.
  - **Jelentése:** A $\mu_A(x)$ érték megadja, hogy az x elem *milyen mértékben* tartozik az A halmazhoz (tagsági fok).
  - **Értékek interpretációja:**
    - 0: Egyáltalán nem tagja.
    - 1: Teljes mértékben tagja.
    - $0 < \mu < 1$: Részleges tagság (átmenet).
- **Típusai:** Háromszög, trapéz, Gauss-görbe (harang) alakú függvények a leggyakoribbak.

59.3 3. Nyelvi változó (Linguistic Variable)

A nyelvi változó a fuzzy logika legmagasabb absztrakciós szintje, amely lehetővé teszi a számításokat szavak (nyelvi kifejezések) segítségével, az emberi gondolkodáshoz hasonlóan.

- **Definíció:** Olyan változó, amelynek értékei nem számok, hanem egy természetes vagy mesterséges nyelv szavai, mondatai.
- **Példa:**

- **Változó neve:** „Sebesség”
- **Értékei (Termek):** „Lassú”, „Közepes”, „Gyors”, „Nagyon gyors”.

- **Formális definíció (Ötös):** $(x, T(x), U, G, M)$
  1. x: A változó neve (pl. Hőmérséklet).
  2. $T(x)$: A nyelvi értékek (termek) halmaza (pl. {Hideg, Langyos, Meleg}).
  3. U: Az alaphalmaz (univerzum), ahol a fizikai mérés történik (pl. $0..100°C$).
  4. G: Szintaktikai szabály (nyelvitan), amely generálja a lehetséges értékeket.
  5. M: Szemantikai szabály, amely minden nyelvi értékhez hozzárendel egy *fuzzy halmazt* (tagsági függvényt) az U univerzumon.

59.4 Összefüggés a fogalmak között

```
# Pszeudokód példa a fogalmak kapcsolatára

# 1. Univerzum (U): A fizikai mennyiség (pl. bemeneti hőmérséklet)
input_homersaklet = 25.5

# 2. Nyelvi változó: "Hőmérséklet"
# Nyelvi érték (Term): "Meleg"

# 3. Tagsági függvény (mu_Meleg): A "Meleg" fogalom definíciója
def membership_meleg(x):
    if x < 20: return 0.0
    if x > 30: return 1.0
    return (x - 20) / 10.0 # Lineáris átmenet (Trapéz/Háromszög széle)

# Eredmény: Tagsági fok (0 és 1 között)
tagsagi_fok = membership_meleg(input_homersaklet)
# tagsagi_fok = 0.55 -> "55%-ban meleg"
```

60 Ismertesse példával azt a szituációt, amikor egy fuzzy partíció lefedi az alaphalmazt! Adja meg a szöveges definíciót is!

A fuzzy rendszerek tervezésekor kritikus szempont, hogy a bemeneti változók teljes tartományát (univerzumát) lefedjük szabályokkal. Ezt biztosítja a helyes fuzzy partíció.

60.1 Szöveges és Formális Definíció

Egy adott X alaphalmaz (univerzum) $A_1, A_2, \ldots, A_n$ fuzzy halmazai akkor alkotnak **teljes lefedést (partíciót)**, ha az alaphalmaz bármely pontjára igaz, hogy a fuzzy halmazok tagsági értékeinek összege pontosan 1.

Ezt a szakirodalom gyakran **Ruspini-partíciónak** vagy egységfelbontásnak nevezi.

$$\forall x \in X : \sum_{i=1}^{n} \mu_{A_i}(x) = 1$$

Jelentése a gyakorlatban:

- **Nincs lyuk:** Nincs olyan pontja az alaphalmaznak, amelyre ne lenne értelmezve legalább egy szabály (a rendszer minden bemenetre tud reagálni).
- **Átmenet:** Ahol az egyik halmaz tagsági értéke csökken, ott egy másiké ugyanolyan mértékben nő.

60.2 Gyakorlati Példa: Víz hőmérséklet szabályozás

Tekintsük egy bojler szabályozását, ahol az X alaphalmaz a víz hőmérséklete 0°C és 100°C között.

- **Univerzum:** $X = [0, 100]$
- **Nyelvi értékek (Fuzzy halmazok):**
  1. **Hideg (H):** 0-nál 1, 50-nél 0.
  2. **Langyos (L):** 0-nál 0, 50-nél 1, 100-nál 0.
  3. **Forró (F):** 50-nél 0, 100-nál 1.

A lefedés ellenőrzése egy adott pontban: Legyen a víz hőmérséklete $x = 25$°C. A háromszög alakú tagsági függvények alapján:

- $\mu_{\text{Hideg}}(25) = 0.5$ (Félig hideg)
- $\mu_{\text{Langyos}}(25) = 0.5$ (Félig langyos)
- $\mu_{\text{Forró}}(25) = 0.0$ (Egyáltalán nem forró)

$$\text{Összeg} = 0.5 + 0.5 + 0.0 = 1.0$$

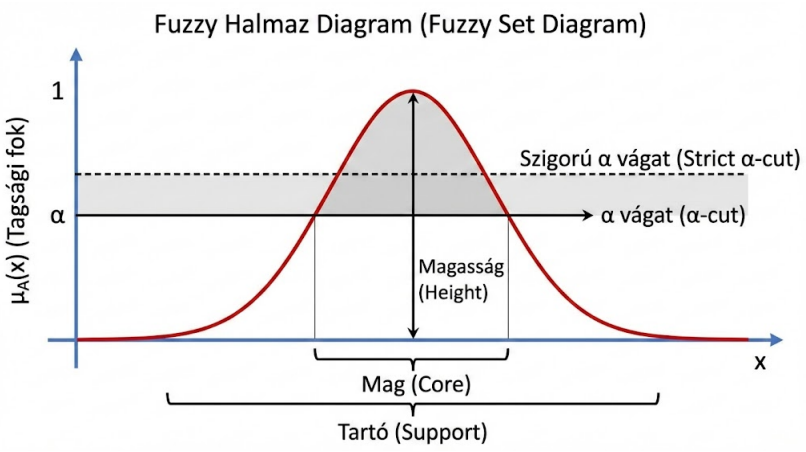
Mivel ez az egyenlőség a $[0, 100]$ intervallum minden pontjára teljesül (a háromszögek "keresztezik" egymást 0.5-nél), a partíció **lefedi** az alaphalmazt.

61 Ismertesse ábrával a „mag”, „tartó”, „α vágat”, „szigorú α vágat” és „magasság” fogalmakat a fuzzy halmazok esetén!

A fuzzy halmazok jellemzésére és a hagyományos (crisp) halmazokkal való összekapcsolására speciális fogalmakat használunk. Ezek a fogalmak a tagsági függvény ($\mu_A(x)$) különböző tulajdonságait írják le.

61.1 Szemléltető Ábra

Az alábbi ábra egy trapéz alakú fuzzy halmazon mutatja be a definíciókat.



1. ábra: Fuzzy halmaz jellemzői: magasság, tartó, mag, α-vágat és szigorú α-vágat

61.2 Definíciók

Legyen X az alaphalmaz (univerzum) és A egy ezen értelmezett fuzzy halmaz.

- **Magasság (Height):** A tagsági függvény által felvett legnagyobb érték (szuprémum).

$$\text{hgt}(A) = \sup_{x \in X} \mu_A(x)$$

  - Ha $\text{hgt}(A) = 1$, a fuzzy halmazt **normálnak** nevezzük.
  - Ha $\text{hgt}(A) < 1$, a fuzzy halmaz **szubnormál**.

- **Tartó (Support):** Az alaphalmaz azon elemeinek halmaza (hagyományos halmaz), amelyek tagsági foka *nagyobb, mint nulla*.

$$\text{supp}(A) = \{x \in X \mid \mu_A(x) > 0\}$$

  Gyakorlatilag ez a fuzzy halmaz "szélessége" az alján.

- **Mag (Core):** Az alaphalmaz azon elemeinek halmaza, amelyek *teljes mértékben* (1-es értékkel) tartoznak a fuzzy halmazhoz.

$$\text{core}(A) = \{x \in X \mid \mu_A(x) = 1\}$$

- **α-vágat (α-cut):** Egy $\alpha \in [0, 1]$ szinten vett vízszintes metszet. Az eredmény egy olyan hagyományos halmaz, amely tartalmazza mindazon elemeket, amelyek tagsági foka eléri vagy meghaladja az α szintet.

$$A_\alpha = \{x \in X \mid \mu_A(x) \geq \alpha\}$$

  Megjegyzés: A mag nem más, mint az 1-vágat (A_1).

- **Szigorú α-vágat (Strong α-cut):** Hasonló a sima vágathoz, de itt szigorú egyenlőtlenséget használunk (az éppen α értékű elemek nem kerülnek bele).

$$A_{\bar{\alpha}} = \{x \in X \mid \mu_A(x) > \alpha\}$$

Megjegyzés: A tartó nem más, mint a szigorú 0-vágat ($A_{\bar{0}}$).

62 Ismertesse a Zadeh szerinti „s-norma”, „t-norma” és komplemens képzését fuzzy halmazoknál!

A fuzzy logikában a hagyományos halmazműveletek (komplemens, metszet, unió) általánosításra kerülnek, hogy értelmezhetők legyenek a $[0, 1]$ intervallumon mozgó tagsági értékekre. Lotfi A. Zadeh eredeti definíciói alkotják a „standard” fuzzy műveleteket.

62.1 1. Fuzzy Komplemens (Tagadás)

A klasszikus logikai „NEM” ($\neg$) művelet kiterjesztése. Zadeh az úgynevezett **standard komplemenst** vezette be.

- **Jelölése:** $\overline{A}$ vagy $\neg A$.
- **Működési elv:** Minél inkább tagja egy elem a halmaznak, annál kevésbé tagja a komplemensének (szimmetria a 0.5 értékre).
- **Képlet:**
$$\mu_{\overline{A}}(x) = 1 - \mu_A(x)$$
- **Axiómák (követelmények):**
  - *Peremfeltételek:* $c(0) = 1$ és $c(1) = 0$.
  - *Monotonitás:* Ha $\mu_A(x)$ nő, a komplemens csökken.
  - *Involúció:* A tagadás tagadása az eredeti érték ($\neg(\neg x) = x$).

62.2 2. T-norma (Fuzzy Metszet / ÉS)

A „t-norma” (Triangular norm) a klasszikus „ÉS” (metszet, $\cap$) művelet általánosított, axiomatikus leírása.

- **Zadeh definíciója (Standard metszet):** Zadeh a **MINIMUM** operátort javasolta a fuzzy metszet képzésére.
$$\mu_{A\cap B}(x) = \min(\mu_A(x), \mu_B(x))$$
Ez a létező legnagyobb (legszigorúbb) t-norma.
- **Szemléletes jelentése:** Egy lánc olyan erős, mint a leggyengébb láncszeme. Ha két feltételnek egyszerre kell teljesülnie, az eredő igazságértéket a kevésbé teljesülő határozza meg.
- **Általános T-norma axiómák:** Egy $T(x, y) : [0, 1]^2 \to [0, 1]$ függvény t-norma, ha:
  - *Kommutatív:* $T(a, b) = T(b, a)$
  - *Asszociatív:* $T(a, T(b, c)) = T(T(a, b), c)$
  - *Monoton növekvő:* Ha a bemenetek nőnek, az eredmény nem csökkenhet.
  - *Peremfeltétel (1 az egységelem):* $T(a, 1) = a$

62.3 3. S-norma (Fuzzy Unió / VAGY)

Az „s-norma” (Triangular conorm / T-conorm) a klasszikus „VAGY” (unió, $\cup$) művelet általánosítása.

- **Zadeh definíciója (Standard unió):** Zadeh a **MAXIMUM** operátort javasolta a fuzzy unió képzésére.
$$\mu_{A\cup B}(x) = \max(\mu_A(x), \mu_B(x))$$
Ez a létező legkisebb s-norma.

- **Szemléletes jelentése:** Ha két feltétel közül elég az egyiknek teljesülnie, az eredő igazságértéket a jobban teljesülő határozza meg.
- **Általános S-norma axiómák:** Egy $S(x, y)$ függvény s-norma, ha kommutatív, asszociatív, monoton, és:
  - *Peremfeltétel (0 az egységelem):* $S(a, 0) = a$

62.4 Összefüggés: De Morgan azonosságok

A Zadeh-féle operátorok (Min, Max, $1-x$) konzisztensek egymással, azaz teljesítik a De Morgan törvényeket, ahogy a klasszikus logika is:

$$\neg(A \cap B) = \neg A \cup \neg B$$
$$1 - \min(a, b) = \max(1 - a, 1 - b)$$

62.5 Implementációs példa

Az alábbi kód bemutatja a Zadeh-féle operátorok működését:

```
def fuzzy_operations_zadeh(mu_A, mu_B):
    # 1. Komplemens (NOT)
    not_A = 1.0 - mu_A

    # 2. T-norma (Metszet / AND) -> MINIMUM
    intersection = min(mu_A, mu_B)

    # 3. S-norma (Unió / OR) -> MAXIMUM
    union = max(mu_A, mu_B)

    return not_A, intersection, union

# Példa értékek
a = 0.7
b = 0.4

res = fuzzy_operations_zadeh(a, b)
# Eredmény:
# NOT A: 0.3
# AND:   0.4 (mert 0.4 < 0.7)
# OR:    0.7 (mert 0.7 > 0.4)
```

63 Ismertesse, hogy mikor alkalmazható egy t-norma, s-norma, komplemens definicióit tartalmazó szabályrendszer fuzzy halmazműveletekhez! Mit alkotnak ilyenkor a szabályrendszer elemei?

A fuzzy logikában a különböző műveleteket (metszet, unió, negáció) nem választhatjuk meg egymástól függetlenül, ha konzisztens matematikai rendszert szeretnénk építeni. A három operátornak összhangban kell lennie egymással.

63.1 Alkalmazhatóság feltétele: A De Morgan Hármas

Egy t-norma (T), s-norma (S) és komplemens (c) definícióit tartalmazó szabályrendszer akkor alkalmazható helyesen fuzzy halmazműveletekhez, ha azok kielégítik a **dualitás elvét**, azaz egymás duális párjai a megadott negációra nézve.

Ezt a rendszert **De Morgan Hármasnak** (De Morgan Triplet) nevezzük.

A rendszer akkor konzisztens, ha teljesülnek az általánosított De Morgan azonosságok minden $x, y \in [0, 1]$ esetén:

$$c(S(x, y)) = T(c(x), c(y))$$
$$c(T(x, y)) = S(c(x), c(y))$$

Ha a standard komplemenst használjuk ($c(x) = 1 - x$), akkor a feltétel egyszerűsödik:

$$1 - S(x, y) = T(1 - x, 1 - y)$$

Példák érvényes hármasokra:

- **Zadeh (Standard):** Min, Max, $1 - x$.
- **Szorzat (Probabilisztikus):** xy, $x + y - xy$, $1 - x$.
- **Lukasiewicz (Korlátos):** $\max(0, x + y - 1)$, $\min(1, x + y)$, $1 - x$.

63.2 Mit alkotnak az elemek?

Ha a fenti feltételek teljesülnek, a $([0, 1], T, S, c)$ struktúra egy **De Morgan Algebrát** alkot.

Ez az algebra hasonlít a klasszikus Boole-algebrára, de **két fontos axióma NEM teljesül** benne általánosan:

1. **A harmadik kizárásának elve (Law of Excluded Middle):** A klasszikus logikában $A \cup \neg A = 1$. A fuzzy logikában általában:
   $$S(x, c(x)) \neq 1$$
   (Pl. Zadeh operátoroknál $0.5 \cup 0.5 = 0.5$).
2. **Az ellentmondásmentesség elve (Law of Contradiction):** A klasszikus logikában $A \cap \neg A = 0$. A fuzzy logikában általában:
   $$T(x, c(x)) \neq 0$$

Következmény: A fuzzy logika matematikailag egy disztributív (általában), De Morgan algebra a $[0, 1]$ intervallumon, amely a crisp (0 vagy 1) értékek esetén Boole-algebrává degenerálódik.

```
1 // Példa: A Lukasiewicz hármas implementációja (De Morgan Triplet)
2 #include <algorithm>
3 #include <iostream>
4
```

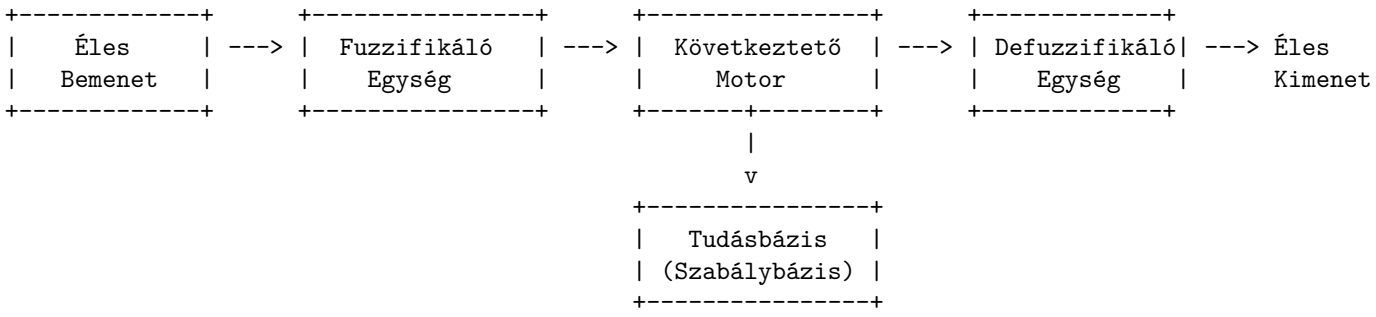
```cpp
// Komplemens: 1 - x
float c(float x) { return 1.0f - x; }

// T-norma (Lukasiewicz): max(0, a + b - 1)
float t_norm(float a, float b) { return std::max(0.0f, a + b - 1.0f); }

// S-norma (Lukasiewicz): min(1, a + b)
float s_norm(float a, float b) { return std::min(1.0f, a + b); }

bool check_de_morgan(float a, float b) {
    // Ellenőrzés: NOT (A OR B) == (NOT A) AND (NOT B)
    float left_side = c(s_norm(a, b));
    float right_side = t_norm(c(a), c(b));

    // Lebegőpontos összehasonlításnál epszilon kellene, de elvben egyenlőek
    return left_side == right_side;
}
```

64 Ismertesse a fuzzy rendszerek általános blokkvázlatát!

A fuzzy következtető rendszerek (Fuzzy Inference System – FIS) célja, hogy éles (crisp) mérési adatokból nyelvi szabályok alapján éles beavatkozó jelet állítsanak elő. A rendszer négy fő komponensből épül fel.

64.1 A rendszer felépítése (Strukturális ábra)

A fuzzy vezérlő adatfolyam-modellje az alábbiak szerint írható le:



64.2 1. Fuzzifikáló egység (Fuzzification Interface)

Ez a modul végzi az átjárást a fizikai világ és a fuzzy logika között.

- **Feladata:** A bemenetről érkező konkrét számértéket (pl. 25°C) átalakítja nyelvi értékekhez tartozó tagsági fokokká.
- **Működése:** A bemeneti tagsági függvényekre (pl. "Hideg", "Meleg") vetíti a bemeneti jelet.
- **Eredménye:** Fuzzy halmazok (tagsági értékek halmaza a $[0, 1]$ intervallumon).

64.3 2. Tudásbázis (Knowledge Base)

A rendszer "agya", amely az alkalmazásspecifikus ismereteket tárolja. Két részből áll:

- **Adatbázis (Database):** Tartalmazza a használt nyelvi változók definícióit és a tagsági függvények paramétereit (formáját, elhelyezkedését).
- **Szabálybázis (Rule Base):** A szakértői tudást leíró IF-THEN (Ha... akkor...) szabályok gyűjteménye.
  - Pl.: *"HA a hőmérséklet magas ÉS a nyomás alacsony, AKKOR a szelep legyen félig nyitva."*

64.4 3. Következtető motor (Inference Engine)

Ez a komponens hajtja végre a fuzzy logikai műveleteket a szabályok alapján.

- **Illesztés:** Meghatározza, hogy az aktuális bemenetekre mely szabályok vonatkoznak (firing strength).
- **Kiértékelés:** A t-normák és s-normák segítségével kombinálja a szabályok feltételeit és levonja a következtetést az egyes szabályokra.
- **Aggregáció:** Az összes aktív szabály részeredményét (a kimeneti fuzzy halmazokat) egyetlen eredő fuzzy halmazzá egyesíti.

64.5 4. Defuzzifikáló egység (Defuzzification Interface)

A fuzzy eredményt visszaalakítja a fizikai világ számára értelmezhető jellé.

- **Feladata:** Az aggregált (bonyolult alakú) fuzzy halmazból egyetlen konkrét számértéket (skalárt) állít elő, amely a legjobban reprezentálja a következtetést.
- **Módszerei:**
  - **COG (Center of Gravity):** Súlypontszámítás (a leggyakoribb).
  - **MOM (Mean of Maxima):** A maximumhelyek átlaga.

65 Ismertesse példával a fuzzy következtető módszer működését!

A fuzzy következtetés (Inference) folyamatát a legszemléletesebben a **Mamdani-típusú** módszerrel lehet bemutatni. A példában egy egyszerű **Ventilátor Szabályozót** valósítunk meg, ahol a bemenet a hőmérséklet, a kimenet a ventilátor fordulatszáma.

65.1 A példa paraméterei

- **Bemenet (x):** Hőmérséklet ($0 \dots 40°\text{C}$).
- **Kimenet (y):** Fordulatszám ($0 \dots 1000$ RPM).
- **Szabálybázis (2 szabály):**
  1. **R1:** HA Hőmérséklet = *Hideg*, AKKOR Fordulat = *Lassú*.
  2. **R2:** HA Hőmérséklet = *Meleg*, AKKOR Fordulat = *Gyors*.
- **Aktuális mérés (Crisp input):** $x_0 = 28°\text{C}$.

65.2 1. Lépés: Fuzzifikálás (Fuzzification)

A konkrét mért értéket (28°C) levetítjük a bemeneti tagsági függvényekre, hogy megkapjuk a tagsági fokokat (μ).

- A 28°C már inkább meleg, de kicsit még hidegnek is számíthat (az átfedés miatt).
- Leolvasott értékek:
  - $\mu_{Hideg}(28) = 0.2$ (20%-ban tartozik a Hideg halmazba).
  - $\mu_{Meleg}(28) = 0.8$ (80%-ban tartozik a Meleg halmazba).

65.3 2. Lépés: Kiértékelés (Inference / Implication)

Meghatározzuk a szabályok "tüzelési erősségét" (activation strength), és ezt alkalmazzuk a kimeneti halmazokra.

- **R1 szabály (Hideg → Lassú):**
  - A szabály feltétele 0.2-es erősséggel teljesült.
  - A kimeneti *Lassú* fuzzy halmazt **levágjuk (csonkoljuk)** a 0.2-es magasságnál (Minimum operátor).
  - Eredmény: Egy 0.2 magas trapéz (a *Lassú* halmaz alja).
- **R2 szabály (Meleg → Gyors):**
  - A szabály feltétele 0.8-as erősséggel teljesült.
  - A kimeneti *Gyors* fuzzy halmazt **levágjuk** a 0.8-as magasságnál.
  - Eredmény: Egy 0.8 magas trapéz (a *Gyors* halmaz nagy része).

65.4 3. Lépés: Aggregáció (Aggregation)

A szabályok részeredményeit egyesítjük egyetlen eredő fuzzy halmazzá.

- A két csonkolt alakzatot (a kicsi *Lassú*-t és a nagy *Gyors*-at) "egymásra rakjuk" az **Unió (Maximum)** művelettel.
- Az eredmény egy komplex, szabálytalan alakzat, amely a kimeneti univerzum felett helyezkedik el.

65.5 4. Lépés: Defuzzifikálás (Defuzzification)

Az aggregált alakzatból egyetlen konkrét számot kell nyernünk a ventilátor vezérléséhez.

- **Módszer:** Súlypontszámítás (Center of Gravity - COG).
- Megkeressük az eredő síkidom súlypontjának x koordinátáját.
- Mivel a *Gyors* halmaz dominál (0.8 vs 0.2), a súlypont a magasabb tartomány felé tolódik.
- **Eredmény:** $y_{out} \approx 750$ RPM.

65.6 Programkód (Szimuláció)

```
# Pszeudokód a Mamdani következtetésre

# 1. Bemeneti mérés
x_temp = 28

# 2. Fuzzifikálás (Tagsági függvények lekérdezése)
# Feltételezve, hogy definiált függvények
mu_hideg = membership_hideg(x_temp) # 0.2
mu_meleg = membership_meleg(x_temp) # 0.8

# 3. Kiértékelés (Implikáció - MIN operátor)
# A kimeneti halmazok "levágása"
rule1_shape = min(mu_hideg, output_set_lassu) # Lassú halmaz max 0.2 magasan
rule2_shape = min(mu_meleg, output_set_gyors) # Gyors halmaz max 0.8 magasan

# 4. Aggregáció (Unió - MAX operátor)
final_fuzzy_shape = max(rule1_shape, rule2_shape)

# 5. Defuzzifikálás (Súlypont)
# Integrálás az y tengely mentén
output_rpm = center_of_gravity(final_fuzzy_shape)

print(f"Ventilator fordulat: {output_rpm} RPM")
```

66 Ismertessen defuzzifikációs módszereket!

A defuzzifikáció a fuzzy következtetési folyamat utolsó lépése. A szabályok kiértékelése és aggregálása után kapott *eredő fuzzy halmazt* (amely egy függvény a $[0, 1]$ intervallumon) átalakítja egyetlen konkrét, **éles (crisp) számértékké**. Erre azért van szükség, mert a fizikai beavatkozók (pl. motor feszültsége, szelep nyitása) konkrét számokat várnak.

66.1 1. Súlypont módszer (Center of Gravity - COG)

Ez a legelterjedtebb és leggyakrabban használt módszer (pl. Mamdani rendszereknél).

- **Elve:** Az aggregált fuzzy halmaz alatti terület geometriai súlypontjának x koordinátáját határozza meg.
- **Matematikai formula (folytonos eset):**

$$y_{COG} = \frac{\int_X x \cdot \mu(x)\, dx}{\int_X \mu(x)\, dx}$$

- **Diszkrét eset (számítógépes megvalósítás):**

$$y_{COG} \approx \frac{\sum_{i=1}^{n} x_i \cdot \mu(x_i)}{\sum_{i=1}^{n} \mu(x_i)}$$

- **Előnye:** Folytonos, sima átmenetet biztosít a kimeneten. Ha a bemenet kicsit változik, a kimenet is csak kicsit fog változni.
- **Hátránya:** Számításigényes (integrálás vagy sűrű mintavételezés kell).

66.2 2. Maximum középérték módszer (Mean of Maxima - MOM)

Ez a módszer csak azokat az értékeket veszi figyelembe, ahol a tagsági függvény a maximumát veszi fel.

- **Elve:** Megkeresi az eredő halmaz maximumát (pl. a platót). Ha több ilyen pont van (vagy egy szakasz), akkor ezek számtani közepét veszi.
- **Formula:**

$$y_{MOM} = \frac{\int_{x \in M} x\, dx}{\int_{x \in M} dx}$$

  ahol $M = \{x \in X \mid \mu(x) = \text{hgt}(A)\}$ a maximális helyek halmaza.
- **Előnye:** Gyorsabb, mint a COG.
- **Hátránya:** Nem folytonos. A kimenet ugrálhat (pl. ha két távoli csúcs között billeg a maximum, az átlag hirtelen változhat). Inkább alakfelismerésnél (klasszifikáció) használják, szabályozásnál ritkán.

66.3 3. Egyéb maximum-alapú módszerek

Ha a maximális tagsági érték nem egy pontban, hanem egy tartományon (platón) jelentkezik:

- **First of Maxima (FOM):** A maximális értékek közül a legkisebb x koordinátájút választja.
- **Last of Maxima (LOM):** A maximális értékek közül a legnagyobb x koordinátájút választja.

66.4 4. Összegsúlypont módszer (Center of Sums - COS)

Hasonló a COG-hoz, de a halmazok aggregálásánál nem az Uniót (maximum), hanem az összeadást használja.

- **Elve:** A szabályokból kapott rész-halmazok területeit összeadja (az átfedéseket duplán számolja), és ennek számolja a súlypontját.
- **Előnye:** Gyorsabban számolható, mert a részhalmazok súlypontjai előre kiszámolhatók és tárolhatók.

66.5 Összehasonlító példa kódban

```
import numpy as np

def defuzzify_example(x_axis, mu_values):
    # 1. Súlypont (COG)
    # Számláló: x * mu(x) összege
    numerator = np.sum(x_axis * mu_values)
    # Nevező: mu(x) összege (terület)
    denominator = np.sum(mu_values)

    cog = numerator / denominator if denominator != 0 else 0

    # 2. Mean of Maxima (MOM)
    # Megkeressük a legnagyobb tagsági értéket
    max_val = np.max(mu_values)
    # Megkeressük az összes indexet, ahol ez az érték szerepel
    indices = np.where(mu_values == max_val)[0]
    # Vesszük az ezekhez tartozó x értékek átlagát
    mom = np.mean(x_axis[indices])

    return cog, mom

# Példa: Egy "M" alakú eloszlás (két csúcs)
x = np.array([10, 20, 30, 40, 50])
mu = np.array([0.2, 1.0, 0.4, 0.9, 0.1])
# Itt a max 1.0 (20-nál), a második csúcs 0.9 (40-nél)

cog_res, mom_res = defuzzify_example(x, mu)

# Eredmény:
# MOM: 20 (Csak a legmagasabb csúcsot nézi)
# COG: ~28 (A 40-nél lévő nagy tömeg "elhúzza" a súlypontot jobbra)
```

67 Ismertesse az aggregációs operátorok definícióját, és 5 axiómáját!

A fuzzy rendszerekben és döntéstámogatásban az aggregációs operátorok feladata, hogy több bemeneti értékből (pl. több szabály következtetéséből vagy több kritérium értékeléséből) egyetlen összevont értéket állítsanak elő.

67.1 Definíció

Az $h : [0,1]^n \to [0,1]$ leképezést aggregációs operátornak nevezzük, ha az több (n darab), a $[0,1]$ zárt intervallumba eső bemenethez rendel hozzá egyetlen, szintén a $[0,1]$ intervallumba eső kimeneti értéket.

Formálisan: $y = h(x_1, x_2, \ldots, x_n)$.

67.2 Az 5 alapvető axióma (Tulajdonság)

A szakirodalom általában az alábbi öt tulajdonságot (axiómát) várja el egy általános célú, jól viselkedő aggregációs operátortól:

1. **Peremfeltételek (Boundary Conditions):** Az operátornak meg kell őriznie az univerzum szélsőértékeit. Ha minden bemenet a minimum, az eredmény is minimum; ha minden bemenet maximum, az eredmény is maximum.
$$h(0,0,\ldots,0) = 0$$
$$h(1,1,\ldots,1) = 1$$

2. **Monotonitás (Monotonicity):** Az operátor nem csökkenő függvény a változói szerint. Ha bármelyik bemeneti érték növekszik (miközben a többi változatlan), a kimeneti érték nem csökkenhet.
$$\text{Ha } x_i \leq y_i \text{ minden } i\text{-re, akkor } h(\mathbf{x}) \leq h(\mathbf{y})$$

3. **Folytonosság (Continuity):** Az operátor legyen folytonos függvény a teljes értelmezési tartományon. Ez biztosítja a rendszer stabilitását: a bemenetek kis megváltozása csak kis változást okozhat a kimeneten, nincsenek ugrásszerű változások.

4. **Szimmetria vagy Kommutativitás (Symmetry):** A bemeneti változók sorrendje nem befolyásolhatja az eredményt (a bemenetek egyenrangúak).
$$h(x_1, x_2, \ldots, x_n) = h(x_{\pi(1)}, x_{\pi(2)}, \ldots, x_{\pi(n)})$$
ahol π az indexek egy tetszőleges permutációja.

5. **Idempotencia (Idempotency):** Ha az összes bemenet ugyanazt a c értéket veszi fel, akkor az aggregált értéknek is pontosan c-nek kell lennie.
$$h(c,c,\ldots,c) = c$$
Megjegyzés: Ez a tulajdonság jellemzően az átlagoló (averaging) operátorokra igaz, a t-normákra és s-normákra általában nem.

67.3 Osztályozás az idempotencia alapján

Az 5. axiómához való viszony alapján az operátorok lehetnek:

- **Átlagoló (Averaging):** Teljesítik az idempotenciát (pl. számtani közép).
- **Konjunktív (And-like):** $h(c,\ldots,c) \leq c$ (pl. t-normák, min).
- **Diszjunktív (Or-like):** $h(c,\ldots,c) \geq c$ (pl. s-normák, max).

68 Ismertesse az általános hatványközép operátort, és a paraméter speciális eseteiben elnevezett értékeit!

Az általános hatványközép (más néven Hölder-közép) egy olyan paraméterezhető aggregációs operátor család, amely a p paraméter változtatásával képes lefedni a teljes skálát a minimumtól a maximumig, magában foglalva a klasszikus középértékeket is.

68.1 Definíció

Legyen $x_1, x_2, \ldots, x_n$ a bemeneti értékek halmaza, ahol $x_i \in [0, 1]$ (vagy pozitív valós számok). Az általános hatványközép operátort a $p \in \boldsymbol{R}$ ($p \neq 0$) paraméterrel a következőképpen definiáljuk:

$$M_p(x_1, \ldots, x_n) = \left(\frac{1}{n} \sum_{i=1}^{n} x_i^p \right)^{\frac{1}{p}}$$

68.2 Speciális esetek (Nevezetes közepek)

A p paraméter különböző értékeire (illetve határértékeire) visszakapjuk az ismert középértékeket:

- $p = 1$**: Számtani közép (Arithmetic Mean)**

  $$M_1(\mathbf{x}) = \frac{1}{n} \sum_{i=1}^{n} x_i$$

  Ez a leggyakrabban használt átlagoló operátor.

- $p = 2$**: Négyzetes közép (Quadratic Mean / RMS)**

  $$M_2(\mathbf{x}) = \sqrt{\frac{1}{n} \sum_{i=1}^{n} x_i^2}$$

  Főleg jelfeldolgozásban és fizikában használatos.

- $p \to 0$**: Mértani közép (Geometric Mean)** Bár a képletbe közvetlenül nem helyettesíthető be a 0, L'Hôpital-szabállyal belátható:

  $$\lim_{p \to 0} M_p(\mathbf{x}) = \sqrt[n]{\prod_{i=1}^{n} x_i}$$

  Akkor használatos, ha az arányok fontosabbak a különbségeknél.

- $p = -1$**: Harmonikus közép (Harmonic Mean)**

  $$M_{-1}(\mathbf{x}) = \frac{n}{\sum_{i=1}^{n} \frac{1}{x_i}}$$

  Párhuzamos ellenállások vagy átlagsebesség számításánál fordul elő.

- $p \to \infty$**: Maximum operátor (Max)**

  $$\lim_{p \to \infty} M_p(\mathbf{x}) = \max(x_1, \ldots, x_n)$$

  Ez a legnagyobb "VAGY" jellegű (s-norma) aggregáció ebben a családban.

- $p \to -\infty$**: Minimum operátor (Min)**

  $$\lim_{p \to -\infty} M_p(\mathbf{x}) = \min(x_1, \ldots, x_n)$$

  Ez a legszigorúbb "ÉS" jellegű (t-norma) aggregáció.

68.3 Tulajdonságok

- **Monotonitás a p szerint:** Az operátor értéke a p növelésével monoton nő.

$$\min(\mathbf{x}) \leq M_{-1}(\mathbf{x}) \leq M_0(\mathbf{x}) \leq M_1(\mathbf{x}) \leq \max(\mathbf{x})$$

- **Idempotencia:** Minden p esetén teljesül, hogy $M_p(c, \ldots, c) = c$.
- **Kompenzáció:** Átlagoló operátor lévén az eredmény mindig a legkisebb és legnagyobb bemeneti érték közé esik.

69 Ismertesse az OWA operátort!

Az OWA (Ordered Weighted Averaging – Rendezett Súlyozott Átlag) operátort Ronald R. Yager vezette be 1988-ban. Ez egy olyan aggregációs művelet, amely hidat képez a **Minimum** (ÉS jellegű) és a **Maximum** (VAGY jellegű) operátorok között, lehetővé téve a „szigorúság” finomhangolását.

69.1 Definíció

Az OWA operátor egy $F : \boldsymbol{R}^n \to \boldsymbol{R}$ leképezés, amelyhez tartozik egy $W = [w_1, w_2, \ldots, w_n]$ súlyvektor, ahol:

1. $w_i \in [0, 1]$
2. $\sum_{i=1}^{n} w_i = 1$

A függvény értéke:

$$F(a_1, a_2, \ldots, a_n) = \sum_{j=1}^{n} w_j b_j$$

Ahol b_j az input argumentumok $(a_1, \ldots, a_n)$ **csökkenő sorrendbe rendezett** permutációjának j-edik eleme.

$$b_1 \geq b_2 \geq \cdots \geq b_n$$

69.2 A legfontosabb különbség a Súlyozott Átlaghoz képest

A hagyományos súlyozott átlagnál (WA) a súlyok a konkrét *adatforráshoz* (attribútumhoz) tartoznak. Az OWA operátornál a súlyok a **pozícióhoz (rangsorhoz)** tartoznak.

- **WA:** A w_1 súly mindig az a_1 bemenetet szorozza, függetlenül annak értékétől.
- **OWA:** A w_1 súly mindig a **legnagyobb** bemeneti értéket szorozza, a w_2 a második legnagyobbat, és így tovább.

69.3 Speciális esetek (A súlyvektor függvényében)

A súlyvektor beállításával az operátor viselkedése változik a két szélsőség között:

- **Maximum (VAGY):** $W^* = [1, 0, \ldots, 0]$
  - Csak a legnagyobb elemet veszi figyelembe.
- **Minimum (ÉS):** $W_* = [0, 0, \ldots, 1]$
  - Csak a legkisebb elemet veszi figyelembe.
- **Számtani közép:** $W_{avg} = [\frac{1}{n}, \frac{1}{n}, \ldots, \frac{1}{n}]$
  - Minden elemet egyformán súlyoz.
- **Olimpiai átlag:** A szélsőértékek kizárása (pl. az első és utolsó súly 0, a többi egyenlő).

69.4 Jellemzők: Orness és Diszperzió

Az operátor karakterisztikáját két mérőszámmal írhatjuk le:

1. **Orness (VAGY-szerűség):** Azt méri, mennyire hasonlít az operátor a Maximumhoz (optimista viselkedés).
   - Ha a súlyok az indexek elején tömörülnek → Magas Orness.
   - Ha a súlyok az indexek végén tömörülnek → Alacsony Orness (ÉS-szerű).

2. **Diszperzió (Entrópia):** Azt méri, mennyire használja ki az összes információt.
    - Ha $W = [\frac{1}{n}, \dots]$, a diszperzió maximális.
    - Ha $W = [1, 0, \dots]$, a diszperzió 0.

69.5 Számítási Példa

Legyenek a bemeneti értékek (pl. három bíráló pontszáma): $A = [0.4, \mathbf{0.9}, 0.7]$. Legyen a súlyvektor (optimista hozzáállás): $W = [0.6, 0.3, 0.1]$.

```
# 1. Bemenetek
inputs = [0.4, 0.9, 0.7]

# 2. RENDEZÉS (Ez a kritikus lépés!)
# Csökkenő sorrendbe állítjuk az értékeket:
sorted_inputs = [0.9, 0.7, 0.4]
# (b1=0.9, b2=0.7, b3=0.4)

# 3. Súlyozott összegzés
# w1*b1 + w2*b2 + w3*b3
result = (0.6 * 0.9) + (0.3 * 0.7) + (0.1 * 0.4)

# Számítás: 0.54 + 0.21 + 0.04
# OWA Eredmény = 0.79
```

Látható, hogy az eredmény (0.79) magasabb, mint a számtani átlag (≈ 0.66), mert a legnagyobb érték kapta a legnagyobb súlyt.