

COMPUTERBOOKS
BUDAPEST, 2008



LEKTOR
DR. TAMÁS PÉTER

Bővített, átdolgozott kiadás

Programozunk C++ nyelven! Az ANSI C++ tankönyve

TÓTH BERTALAN
LAPTEVA NATALIA

A könyv készítése során a Szerzők és a Kiadó a legnagyobb gondossággal jártak el. Ennek ellenére hibák előfordulása nem kizárható.
Az ismeretanyag felhasználásának következményeért sem a Kiadó, sem a Szerzők felelősséget nem vállalnak.



Oktatási
Minisztérium

A könyv az Oktatási Minisztérium támogatásával, a Felsőoktatási Pályázatok Irodája által lebonyolított Felsőoktatási Tankönyv- és Szakkönyvtámogatás Pályázat keretében jelent meg.

Minden jog fenntartva. Jelen könyvet, vagy annak részleteit a Kiadó engedélye nélkül bármilyen formában, vagy eszközzel reprodukálni, tárolni, közölni tilos.

© Lapteva Natalia, Tóth Bertalan, 2003, 2005, 2006

Negyedik kiadás, 2007

ISBN: 963 618 346 5

© Kiadó, ComputerBooks Kiadó Kft.
1126 Budapest Tartsay Vilmos u. 12
Telefon: 375-1564; Tel/Fax: 375-3591
E-mail: info@computerbooks.hu
<http://www.computerbooks.hu>
Felelős kiadó: ComputerBooks Kft. Ügyvezetője
Bortóterv: Székely Edit

Tartalomjegyzék

Bevezetés.....	1
I. A C++ MINT EGY JOBB C NYELV	7
1. Első találkozás C++ nyelven írt programmal	9
2. A C++ nyelv alapjai.....	11
2.1 A nyelv jelkészlete	11
2.2 A C++ nyelv azonosítói.....	11
2.3 Konstansok	12
2.3.1 Egész konstansok	13
2.3.2 Karakterkonstansok	14
2.3.3 Lebegőpontos konstansok	15
2.4 Sztringkonstansok (litterálok)	15
2.5 Megjegyzések a programban	16
2.6 Operátorok és írásjelek	16
3. A C++ program szerkezete	17
3.1 Egyetlen modulból felépülő C++ program.....	17
3.2 Több modulból álló C++ program.....	18
4. Alaptípusok, változók és konstansok	21
4.1 A C++ nyelv típusai	21
4.1.1 Típuselőírások, típusmódosítók.....	22
4.1.2 Típusminősítők	23
4.1.3 A felsorolt típus (enum)	24
4.2 Egyszerű változók definiálása	25
4.3 Saját típusok előállítása.....	26
4.4 Konstansok a C++ nyelvben.....	26
4.5 Értékek, címek, mutatók és referenciák	28
4.5.1 Balérték és jobbérték	28
4.5.2 Ismerkedés a mutatóval és a referenciával	28
5. Operátorok és kifejezések.....	33
5.1 Precedencia és asszociativitás	36
5.1.1 Az előbbbségi (precedencia) szabály.....	36
5.1.2 A csoportosítási (asszociativitás) szabály.....	37
5.2 Mellékhatások és a rövidzár kiértékelés.....	37
5.3 Az elsőleges kifejezés operátorai	38
5.4 Aritmetikai operátorok	38

5.5	Összehasonlító és logikai operátorok	39
5.6	Léptető operátorok	41
5.7	Bitműveletek	43
5.7.1	Bitenkénti logikai műveletek	43
5.7.2	Biteltoló műveletek	44
5.8	Értékadó operátorok	45
5.9	Pointerműveletek	47
5.10	A sizeof operátor	48
5.11	A vesso? operátor	49
5.12	A felte?les operátor	49
5.13	Az érvényesség? kör (hatókör) operátor	50
5.14	A new és a delete operátorok használata	51
5.15	Futásiidej? típusazonosítás	54
5.16	Típuskonverziók	55
5.16.1	Implicit típuskonverziók	55
5.16.2	Explicit típusátalakítások	56
5.17	Bővebben a konstansokról	57
6. A C++ nyelv utasításai		59
6.1	Utasítások és blokkok	59
6.2	Az if utasítás	60
6.2.1	Az if-else szerkezet	62
6.2.2	Az else-if szerkezet	63
6.3	A switch utasítás	65
6.4	A ciklusutasítások	67
6.4.1	A while ciklus	68
6.4.2	A for ciklus	69
6.4.3	A do-while ciklus	71
6.5	A break és a continue utasítások	73
6.5.1	A break utasítás	73
6.5.2	A continue utasítás	74
6.6	A goto utasítás	76
6.7	A return utasítás	77
6.8	Kivételek kezelése	77
6.9	Definíciók bevittele az utasításokba	81
7. Származtatott adattípusok		83
7.1	Tömbök, sztringek és mutatók	83
7.1.1	Egydimenziós tömbök	83
7.1.2	Mutatók és a tömbök	87
7.1.3	Karakter sorozatok (sztringek)	89
7.1.4	Többdimenziós tömbök	92
7.1.5	Mutatótömbök, sztringtömbök	94

7.1.6 Dinamikus helyfoglalású tömbök	97
7.2 Felhasználó által definiált adattípusok	101
7.2.1 A <i>struct</i> struktúrátípus	102
7.2.2 A <i>class</i> osztálytípus	111
7.2.3 A <i>union</i> típusú adattípusok	112
7.2.4 A bitmezők alkalmazása	115
7.2.5 Esettanulmány: önhivatkozó struktúrák használata - listaszervezet	118
8. Függvények	123
8.1 Függvények definíciója és deklarációja	123
8.2 A függvények paraméterezése és a függvényérték	126
8.3 A függvényhívás	127
8.4 Különböző típusú paraméterek használata	130
8.4.1 Aritmetikai típusú paraméterek	130
8.4.2 Felhasznált típusú paraméterek	131
8.4.3 Tömbök átadása függvénynek	133
8.4.4 Stringargumentumok	136
8.4.5 A függvény, mint argumentum	138
8.4.6 Alapértelmezés szerinti (default) argumentumok	146
8.4.7 Változó hosszúságú argumentumlista	147
8.4.8 A <i>main()</i> függvény paraméterei és visszatérési értéke	149
8.5 Rekurzív függvények használata	151
8.5.1 A rekurzív függvények csoportosítása	153
8.6 Inline függvények	156
8.7 Függvénynevek átdefiniálása (overloading)	157
8.8 Alternatív függvények (template)	159
8.9 Típusmegörző szerkesztés (type-safe linking)	161
8.10 Esettanulmány: C++ deklarációk értelmezése és készítése	162
8.10.1 C++ deklarációk értelmezése	162
8.10.2 C++ deklarációk készítése	164
9. Névterek és tárolási osztályok	167
9.1 Az azonosítók élettartama	167
9.2 Ervényességi tartomány és a láthatóság	168
9.3 A kapcsolódás	169
9.4 Névterületek	170
9.4.1 Saját névterületek kialakítása és használata	170
9.5 A tárolási osztályok használata	174
9.5.1 Az auto tárolási osztály	175
9.5.2 Az extern tárolási osztály	176
9.5.3 A static tárolási osztály	177
9.5.4 A register tárolási osztály	180

10. Az előfeldolgozó (preprocesszor).....	181
10.1 Állományok beépítése a forrásprogramba.....	182
10.2 Feltehetőes fordítás.....	183
10.3 Makrók használata.....	185
10.3.1 Szimbolikus konstans makrók készítése.....	186
10.3.2 Függvényyszerű makrók készítése.....	186
10.3.3 Előre definiált makrók.....	189
10.4 A #line, az #error és a #pragma direktívák.....	190
II. AZ OBJEKTUM-ORIENTÁLT C++ NYELV.....	191
II. Bevezetés az objektum-orientált C++ nyelvbe.....	193
12. Osztályok.....	197
12.1 Adattagok.....	198
12.2 Tagfüggvények.....	198
12.2.1 Konstans tagfüggvények és a mutable típusmódosító.....	200
12.3 Az osztály tagjainak elérése.....	202
12.3.1 Az osztályok friend mechanizmusa.....	203
12.4 Az osztály objektumai.....	204
12.5 Statikus osztálytagok használata.....	206
12.6 Osztálytagokra mutató pointerek.....	207
13. Konstruktorok és a destruktork.....	209
13.1 Konstruktorok.....	209
13.1.1 A konstruktorok explicit paraméterezése.....	213
13.1.2 Tagobjektumok inicializálása.....	213
13.1.3 Konstans tagok inicializálása.....	214
13.2 Destruktor.....	215
14. Operátorok túlterhelése (operator overloading).....	217
14.1 A léptető (++ , --) operátorok túlterhelése.....	220
14.2 Az értékadó operátor átdefiniálása.....	222
14.3 Függvényyszerű osztályok (functionoidok) készítése.....	223
14.4 Felhasznált által definiált típuskonverzió.....	224
14.5 Az osztályok bővítése input/output műveletekkel.....	225
14.6 A new és a delete operátorok túlterhelése.....	226
14.7 A C++ nyelv bővítése saját típussal.....	227
15. Az öröklés (öröklődés) mechanizmusa.....	231
15.1 A származtatott osztályok.....	231
15.2 Az alaposztály inicializálása.....	235
15.3 Virtuális tagfüggvények – többalakúság (polimorfizmus).....	237
15.3.1 A virtuális függvények újradefiniálása (redefine).....	237
15.3.2 Virtuális destruktork.....	239

15.4 Virtuális alapsztylyok.....	240
15.5 Futás közbeni típusinformációk (RTTI) osztályok esetén	241
15.6 Kivételek és az osztályok	245
16. Általánosított osztályok (templates)	249
16.1 Általánosított osztály definíciója	253
16.2 Specializáció és példányosítás	253
16.3 A sablonosztály „barátai” és statikus adattagjai.....	255
16.4 Érték- és alapértelmezett sablonparaméterek	255
16.5 A typename kulcsszó.....	256
16.6 Összetettebb sablonpélda	257
III. A C++ NYELV KÖNYVTÁRA	261
F1. A szabványos C++ nyelv könyvtárainak áttekintése	263
F2. Adatok bevétele és kivétele	267
F2.1 A C-könyvtár alapvető I/O műveletei (<i>csdio</i>)	267
F2.1.1 Karakterek kiírása és beolvasása	268
F2.1.2 Karaktertörsorozatok kiírása és beolvasása	269
F2.1.3 Formázott adatok bevétele és megjelenítése	269
F2.1.4 Írás sztringbe és olvasás sztringből	276
F2.1.5 Az <i>sídio</i> és <i>sídot</i> adatfolyamok átirányítása	276
F2.2 A C++ alapvető I/O műveletei (<i>íostream</i>).....	277
F2.2.1 Az << és >> műveletek.....	278
F2.2.2 I/O manipulátorok használata	279
F2.2.3 I/O manipulátorok készítése	282
F2.2.4 Írás sztringbe és olvasás sztringből	283
F2.3 Állományok kezelése	284
F2.3.1 Állománytípusok	284
F2.3.2 Szabványos C/C++ fájlkezelés	285
F2.3.3 A szabványos C++ állománykezelés	287
F3. A C-könyvtár legfontosabb elemei	291
F3.1 Karakterek osztályozása és átalakítása	291
F3.2 Konverziós függvények	292
F3.3 Memóriaterületek (pufferek) kezelése	294
F3.4 A dinamikus memóriakezelés függvényei.....	295
F3.5 Matematikai függvények.....	296
F3.6 Dátum- és időkezelő függvények.....	300
F3.7 Rendezés és keresés	303
F3.8 Egyéb hasznos függvények	304
F4. A Szabványos sablonkönyvtár (STL) elemei.....	305
F4.1 Tárolók (konténerek).....	305

F4.2 Iterátorok.....	311
F4.3 Algoritmusok.....	313
F4.4 Műveletobjektumok (<i>functions</i>).....	315
F4.5 Allokátorok (<i>allocators</i>).....	317
F5. Karakterosorozatok kezelése.....	319
F5.1 A C-könyvtár függvényei.....	319
F5.2 A string osztály használata.....	322
Irodalomjegyzék.....	327
Tárgymutató.....	329

Bevezetés

Egy új könyvvel való ismerkedés során két kérdésre kereshetjük a választ. Milyen a könyv felépítése, mit ad nekem ez a mű? A másik, nem kevésbé fontos kérdés, hogy mi is az a könyvben tárgyalt C++ nyelv?

Gondolatok a könyvről

A „*Programozunk C++ nyelven*” című könyv elsősorban azok számára íródott, akik most kezdenek ismerkedni napjaink programozási nyelveinek alapjait képező C++ nyelvel. Nem szűkséges a teljes művet elolvasni ahhoz, hogy a feldolgozott témakört programok írásával mélyítse el az Olvasó. Amennyiben a feldolgozást sikeresnek ítéli meg, továbbbővíthet a következő fejezetre. Az önellenőrzéshez a CD-mellékleten található C++ fejlesztőeszköz, tesztprogram és feladatok nyújtanak segítséget.

A könyv azok számára is értékes szakirodalom lehet, akik ugyan már programokat írnak C++ nyelven, azonban a C++ nyelvnek még nem minden területén mozognak otthonosan. A fejezetek a címükben szereplő témakört teljes részletességgel és példák sokaságával tárják az Olvasó elé.

A könyv felépítése

A könyv felépítése olyan, hogy folyamatosan olvasva meg lehet ismerkedni a C++ nyelvel, valamint a legfontosabb könyvtári függvényekkel és osztályokkal. A C++ teit három részre osztjuk.

Az első, „A C++ *mint egy job* C nyelv” rész a C++ nyelv hagyományos programozást támogató lehetőségeit tárja az Olvasó elé. Ezek az ismeretek elsősorban a nem objektum-orientált programkészítés során hasznosíthatók.

A második, „Az *objektum-orientált C++ nyelv*” című rész hat fejezetben keresztül az objektum-orientált programkészítés alapjaival ismereti meg az Olvasót. A tárgyalás során a részletekben való elmélyülés helyett egy világos áttekintést adunk a témakörrel kapcsolatos fogalmakról és megoldásokról.

A harmadik rész valójában a könyv függeléke, azonban ennek ellenére egy sor hasznos információt tartalmaz a C++ nyelv könyvtári függvényeiről és osztályairól. Többek között bemutatjuk az adatbevitel és adatkivitel hagyományos és objektumos megoldásait, a karaktersorozatok kezelésének eszközeit, valamint a szabványos sablon-könyvtár (*Standard Template Library*) összetevőit.

Mit található a CD-mellékleten?

A könyvhöz csatolt CD-melléklet azon túl, hogy megkíméli az Olvasót a példák unalmas beágyazásától, a kitűzött feladatok megoldását is tartalmazza. Ugyancsak megtalálható itt egy tesztprogram és a *Dev-C++ 5.0 beta 9.2 (4.9.2)* ingyenes C++ fejlesztőcsomag. A *Dev-C++* szabad szoftver, amelyet a *GNU General Public License* alapján lehet terjeszteni. (Ez azt jelenti, hogy szabadon továbbadható és módosítható.)

Mitől más ez a könyv?

Könyvünk megírását évtizedes C/C++ oktatási gyakorlat előzte meg a Budapesti Műszaki Egyetem Gépeszmérnöki Karán és Mérnöktovábbképző Intézetében. Ennek során szerzett tapasztalatok nagyban hozzájárultak könyvünk szerkesztéséhez és tartalmának kialakításához. Ezúton is köszönetet szeretnénk mondani annak a több száz hallgatónak, akik közvetve ugyan, de hozzájárultak könyvünk megjelenéséhez.

Gondolatok a C++ nyelvről

A C++ hatékony, általános célú programozási nyelv. Segítségével kis alkalmazások és nagy programok egyaránt előállíthatók. A C++ hagyományos fejlesztőeszközként és objektum-orientált programozási nyelvként egyaránt használható.

A múlt

A C nyelv alapelemeinek többsége a *Martin Richards* által kifejlesztett *BCPL* (*Basic Combined Programming Language*, 1963) nyelvből származik. Ez a származtatás közvetett módon – a *B* nyelven keresztül – ment végbe. A *B* nyelvet *Ken Thompson* dolgozta ki 1970-ben az *AT&T Bell Laboratories* cégénél, és ezen a nyelven készült el az első UNIX operációs rendszer DEC PDP-7 számítógépre.

A *B* nyelv azonban nem bizonyult elég hatékonynak az új PDP-11 számítógép UNIX operációs rendszerének megírására. Mivel azonban a magas szintű nyelven történő implementációról nem akartak lemondani, *Dennis Ritchie* 1971-ben hozzálatott a *B* nyelv új változatának kidolgozásához, amit *C*-nek neveztek el. A felmerült problémák kiküszöbölésén kívül, *Ritchie* egykezezt a C nyelvbe visszahozni azokat az általános programozási elemeket, amelyek az *Algol-60* nyelv „karcsúsítása” során kikerültek a BCPL és *B* nyelvekből.

Több éven át csak az AT&T UNIX operációs rendszerrel szállított C-fordító képviselte a C nyelv definícióját. Az 1978-ban megjelent, *B. W. Kernighan* és *D. M. Ritchie* által írt „*The C Programming Language*” című könyv szolgált a nem UNIX-alapú C-implementációk kiindulópontjául.

A C nyelv világméretű elterjedése összefügg a mikroszámítógépek megjelenésével. Bár ebben az időben még nem volt szabványa a C nyelvnek, azonban a K&R könyvben leírt függvények használatával lehetett olyan programot írni, amely a különböző mikro- és miniszámítógépeken lefordítva ugyanúgy futott. Megjelent az igény a szabványos C nyelv megalkotására, amely a gombamód szaporodó fordítók és gépek között biztosíthatja a C nyelvű forrásprogramok hordozhatóságát (portabilitását).

A C nyelvet több lépésben szabványosították. Az első (ANSI) szabvány 1983-ban jelent meg, így alapul szolgálhatott a C++ nyelv megformálásához. A ANSI C szabvány végleges változata (ANSI X3.159-1989) hat évi munka után, 1989-ben készült el. A C szabványt 1989-ben revízió alá vették (IOS/IEC), majd 1995-ben kibővítették a széles karakterek (*wchar_t*) használatának lehetőségével.

A C nyelv történetét itt le is zárhatnánk, napjainkban újabb C-fordítók már nem születtek.

A jelen

A C nyelv helyét, szerepét világsszerte a C++ nyelv veszi át, melynek kifejlesztése az AT&T Bell Laboratóriumoknál dolgozó *Bjarne Stroustrup* nevéhez fűződik. Mint ahogy ismeretes, a C nyelvet szintén itt fejlesztették ki a 70-es évek elején. Így nem csoda, hogy a tíz évvel későbbi C++ fejlesztés a C nyelvre épült. A C nyelv ismerete ezért teljesen természetes kiindulópont a C++ nyelv megismeréséhez. *Bjarne Stroustrup* két fő szempontot tartott szem előtt a C++ kidolgozásánál:

1. A C++ nyelv legyen felülíró kompatibilis az eredeti C nyelvvel.
2. A C++ nyelv bővítse ki a C nyelvet a *Simula 67* nyelvben használt osztály-szerkezettel (*class*).

Az osztályszerkezet, amely a C nyelv **struct** adatszerkezetére épült, lehetővé tette az objektum-orientált programozás (OOP) megvalósítását.

A C++ nyelv több szakaszban nyerte el mai formáját. A C++ *Version 1.2* változata terjedt el először a világon (1985). A használata során felvetődött problémák és igények figyelembevételével *Bjarne Stroustrup* kidolgozta a *Version 2.0* nyelvédefiniációt (1988). A jelentős változtatások miatt a régi C++ (1.2) nyelven írt programok általában csak kisebb-nagyobb javítások után fordíthatók le a 2.0-as verziót megvalósító fordítóprogrammal. Az évek folyamán a C++ nyelv újabb definíciói (3.x) jelentek meg, azonban a lényeges újdonságok két nagy csoportba sorolhatók:

- kivételek (*exception*) kezelése,
- paraméterezett típusok, osztályok és függvények használata (*templates*, sablonok).

A C++ szabvány kidolgozásában az ANSI X3J16 és az ISO WG21 bizottságok vettek részt a 90-es évek első felében. Munkájuk eredményeként 1997 novemberében megszületett az ANSI/ISO C++ szabvány, melyre a napjainkban használt legtöbb C++ fordítóprogram épül. A szabványosítás folyamata azonban nem zárult le, hisz a gyorsan változó világunk igényeinek megfelelően állandóan módosítani, bővíteni kell a nyelv definícióját.

A C nyelv szabványához hasonlóan, a C++ szabvány sem rögzíti a nyelv összes részletét egyértelműen, kötelező jelleggel. Bár mindig tartalmaz előírást (*shall have*), azonban több esetben megengedi, hogy a C++ fordító fejlesztői eltejejenek ettől - ezek az ún. "implementáció által definiált" esetek. A könyvünkben több helyen hivatkozunk a szabvány előírásaira, például a különböző típusok értelmezése során. Itt csupán példaként, tekintjük a *main()* függvényre vonatkozó passzusok némelyikét!

"Minden programnak tartalmaznia kell egy globális *main* nevű függvényt, amely kijelöli a program belépési pontját. A *main()* függvény típusa *int* kell legyen, de másfélben a típusa implementáció-függő. A *main()* függvényben megadott *return* utasítás először elvégzi a szokásos blokklezárási műveleteket, majd pedig meghívja a csatlakoztatott fejeletben deklarált *exit()* függvényt, átadva neki az utasításban szereplő kifejezés értékét. Ha *main()* nem tartalmazza a *return* utasítást, akkor automatikusan a *return 0*; hajtódik végre a függvény-blokk végének elérésékor."

A C++ nyelv szabványa alapján a különböző C++ fordítóprogramok eltérő módon kezelhetik a *main()* függvény típusát. Vannak, amelyek a szabvány szavait szigorúan értelmezik, és nem engedik meg az *int* típustól eltérő függvénytípus használatát, míg mások *void*, *bool* stb. típusok alkalmazását is lehetővé teszik. (A *void* típus esetén általában a szabvány *return 0*; előírása érvényesül.)

Az elmondottak alapján, az alábbi formák minden fordítóprogrammal hibázó és figyelmeztetés nélkül lefordíthatók, így ezek alkalmazását javasoljuk:

```
int main() {
    // ...
    // kilépési kód 0
}

int main() {
    // ...
    // kilépési kód 0
    return kif;
}

int main() {
    // ...
    // kilépési kód kif;
    exit(kif);
}
```

Könyvünk születésekor minden C++ fordítóprogram hibázóan nem fordította a *void* típusú *main()* függvényt, ezért a szabvány javaslatával nem tördöve, a példaprogramok többségében ezt használtuk.

Ennek ellenére a szigorúbb C++ fejlesztőeszközökkel is lefordíthatjuk a könyvben szereplő programokat, ha a *main()* előtt álló **void** típust *int*-re cseréljük.

Az alábbi táblázatban összefoglaltuk néhány elterjedt C++ fejlesztőeszköz viszonyu-
lását a **void** típusú *main()* függvényhez.

C++ fejlesztőeszköz	<i>void main()</i> megengedett
Microsoft Visual C++ 6.0	✓
Microsoft Visual Studio.NET 2003	✓
Microsoft Visual Studio 2005	✓
Borland C++ Builder 6	✓
Borland Developer Studio 2006	✓
Bloodshed Dev-C++ 4.9.7.0 (Mingw/GCC 3.2)	✓
Bloodshed Dev-C++ 4.9.9.2. (Mingw/GCC 3.4.2)	<i>'main' must return 'int'</i>

Mielőtt elkezdենek a szabványos C++ nyelv elemeinek részletekbe menő bemuta-
tását, le kell szögeznünk, hogy a C++ nyelv a C nyelv szintakszisára épülő önálló
programozási nyelv. Alapvető eltérés a két nyelv között, hogy amíg a C nem típusos
nyelv, addig a C++ erősen típusos, objektum-orientált nyelv. A C programok többsége
minden további nélkül átvihető C++ rendszerbe. Mivel a C++ nyelv „erősen típusos
nyelv”, a szükséges módosításokat általában a konverziók és a deklarációk területén
kell elvégezni.

(Egy lehetséges) jövő

Napjainkban egyre több olyan programozási nyelv születik, melyek gyökere a C/C++
nyelvekből ered. Ilyen szemszögből tekintve elmondhatjuk, hogy a C++ nyelv átvette
az Algol nyelvtől azt a szerepet, hogy más nyelvek is merítenek belőle.

A jövőben a C++ elsősorban a helyi (kliens) fejlesztések eszköze lesz, míg más terüle-
teken az újabb nyelvek hódítanak. Külön is érdemes kiemelni a hálózatos (Internetes)
megoldásokhoz használt eszközök közül a *Sun* cég által támogatott *Java* nyelvet, és
Microsoft hasonló célú fejlesztőeszközét, a *C#* (*C sharp*) nyelvet. Érdekesképpen
nézzünk néhány példát a rokonság bizonyítására!

Egyre nagyobb jelentőséggel bírnak a kiszolgáló oldali fejlesztések programozási nyelvel, mint a PHP, Perl és a Python, amelyek szintén emlékeztetnek a C++-ra. Az áttekintő jellegetű gondolatok után kezdjük el az C++ nyelvel való ismerkedést!

```
// kozos.cpp - C++
#include <iostream>
using namespace std;

int main() {
    Class1::Main();
}

}

// Class1.cs - C#
using System;
public class Class1 {
    public static void Main(string[] args) {
        for (int i=1; i<23; i++)
            if (i%3==0)
                System.Console.WriteLine(i);
    }
}

// Class1.java - Java
import java.lang.*;
public class Class1 {
    public static void main (String[] args) {
        for (int i=1; i<23; i++)
            if (i%3==0)
                System.out.println(i);
    }
}

// kozos.cpp - C++
#include <iostream>
using namespace std;

int main() {
    for (int i=1; i<23; i++)
        if (i%3==0)
            cout<<i<<endl;
}
```

A C++ mint egy jobb C nyelv

I.

1. Első találkozás C++ nyelven írt programmal

Tekintsük az alábbi egyszerű, C nyelven megírt programot, amely bekér egy szöveget és két számot, majd kiterjti a szöveget és a számok szorzatát!

```
#include <stdio.h>
#include <conio.h>
void main()
{
    char nev[20];
    int a;
    double b;

    printf("Kérem a szöveget: ");
    scanf("%s",nev);
    printf("A=");
    scanf("%d",&a);
    printf("B=");
    scanf("%lf",&b);
    printf("%s : A*B=%lf\n",nev,a*b);
    getch();
}
```

A példában a I/O műveletek elvégzéséhez a szabványos C-könyvtárban található *scanf()* és *printf()* függvényeket használtuk. Ezen függvények nem tekinthetők a C nyelv részének, hiszen csak könyvtári függvények. A C++ nyelv a szabványos I/O műveletek kezelésére szintén tartalmaz kiegészítést, a **cin** és a **cout** adatfolyam (*stream*) objektumok definiálásával, amelyek szintén nem képezik részét a C++ nyelv definíciójának. Ezen osztályok felhasználásával a fenti program szabványos C++ nyelven elkészített változata:

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

int main()
{
    char nev[20];
    int a;
    double b;

    cout << "Kérem a szöveget: ";
    cin >> nev;
    cout << "A=";
    cin >> a;
    cout << "B=";
    cin >> b;
    cout << nev << " : A*B=" << a*b;
    cin.get();
    cin.get();
}
```

Szembeöltő eltérés a C nyelvű programhoz képest, hogy az I/O műveletek használata egyes szerűbbé vált. Nem kell figyelni a megfelelő formátum megadására, illetve a helyes paraméterezésre, mindezt elvégzi helyettünk a fordítóprogram. A C++ szabvány minden könyvtári elemet a közös *std* névtérületen definiál. Emiatt a hagyományos C++ megoldás

```
#include <iostream.h>
```

helyett, a szabványos formát alkalmazzuk a példaprogramokban:

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

A szabványos input elvégzésére a *cin*, míg a szabványos outputként a *cout* adatfolyam-objektumot használjuk. Létezik még egy szabványos hibafolyam is, a *cerr*. Mindhárom objektum definícióját az *IOSTREAM* fejlécben találjuk meg. (A 16-bites kódolási karaktereket tartalmazó szövegek kezelését a fenti objektumok *w* betűvel kezdődő párhajai támogatják: *wcout*, *wcin*, *wcerr*.) Az adatfolyam-objektumok lehetőséget a későbbiekben részletesen tárgyaljuk. Az itt bemutatott szabványos adatbeviteli és adatkiviteli (I/O) műveleteket a példaprogramokban kívánjuk felhasználni.

2. A C++ nyelv alapelemei

A C++ nyelvvel való ismerkedés legelején áttekintjük a C++ programozási nyelv azon alapelemeit – a neveket, a számokat és a karaktereket – amelyekből a C++ program felépül. Az ANSI C++ szabvány nyelvhasználatával élve, ezeket az elemeket *token*-nek nevezzük. A C++ forrásprogram fordításakor a fordítóprogram a nyelv tokenjeit dolgozza fel. (A tokeneket a fordító már nem bontja további részekre.). A C++ nyelv alapelemeihez tartoznak a kulcsszavak, az azonosítók, a konstansok, a sztringlitterálok, az operátorok és az írásjelek.

2.1 A nyelv jelkészlete

A szabványos C++ program készítésekor kettőféle jelkészlettel dolgozunk. Az első jelkészlet azokat a karaktereket tartalmazza, amelyekkel a C++ programot megírjuk:

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_	{		}	~	?	+			
a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w	x	y	z	,	.	:	;	'	"	#	\$	%	&	'	(	)	*

A nem látható karakterek közül ide tartoznak még a szóköz, a vízszintes és függőleges tabulátor, a sorameles és a lapdobás karakterek is, melyek feladata a forrásszöveg tagolása. (Ezeket a karaktereket összefoglaló néven *whitespace* karaktereknek hívjuk.) Azok a karakterek (ANSI, Unicode) melyeket nem tartalmaz a C++ nyelv karakterkészlete szintén szerepelhetnek a programban, de csak megjegyzések és sztringlitterálok (szöveggkonstansok) belsejében.

2.2 A C++ nyelv azonosítói

A C++ nyelvű program bizonyos összetevőire (például változókra, függvényekre, címkekre) névvel hivatkozunk. A nevek (azonosítók, szimbólumok) megfelelő megválasztása lényeges része a program írásának. Az azonosítók hossza általában implementációfüggő – a legtöbb fordító legfeljebb 32-karakteres nevek használatát támogatja. Az azonosító első karaktere betű vagy _ (aláhúzásjel) lehet, míg a második karaktertől kezdődően betűk, számok és aláhúzásjelek válthatják egymást. Az azonosítók elején az aláhúzásjel általában a rendszer által használt, illetve a C++ nyelv bővítését jelentő nevekben szerepel. A legtöbb programozási nyelvtől eltérően a C++ nyelv az azono-

sítkban megkülönbözteti a kis és a nagybetűket. Ezért az alábbi nevek egymástól függetlenül használhatók a programban (nem azonosak):

Alma, ALMA

Elterjedt konvenció, hogy kisbetűvel írjuk a C++ azonosítókat és csupa nagybetűvel az előfordító által használt neveket (makrókat).

byte, DEBUG, FALSE

Az értelmes szavakból összeállított azonosítókban az egyes szavakat általában nagybetűvel kezdjük:

Felsosarok, Xkoordinata

Bizonyos azonosítók speciális jelentést hordoznak. Ezeket a neveket foglalt szavaknak vagy kulcsszavaknak nevezzük. A foglalt szavakat a programban csak a hozzájuk rendelt értelmezésnek megfelelően lehet használni. A kulcsszavakat nem lehet átdefinálni, mi, új jelentéssel ellátni. Az alábbi táblázatban összefoglaltuk az ANSI C++ nyelv kulcsszavait:

asm	do	if	return	try
auto	double	inline	short	typedef
bool	dynamic_cast	int	signed	typeid
break	else	long	sizeof	typename
case	enum	mutable	static	union
catch	explicit	namespace	static_cast	unsigned
char	export	new	struct	using
class	extern	operator	switch	virtual
const	false	private	template	void
const_cast	float	protected	this	volatile
continue	for	public	throw	wchar_t
default	friend	register	true	while
delete	goto	reinterpret_cast		

A legtöbb fordítóprogram kibővíti a szabványos kulcsszavakat saját jelentéssel bíró szavakkal. Erre a C++ szabvány a két aláhúzással használataát javasolja, például:

`__try, __property, __published, __int64`

2.3 Konstansok

A C++ nyelv megkülönbözteti a numerikus és a szöveges konstansértékeket. A konstansok alatt mindig valamiféle számot értünk, míg a szöveges konstansokat sztring-iteráinak hívjuk. A konstans értékek ilyen megkülönböztetését a tárolási és felhasználási módjuk indokolja.

A C++ nyelvben karakteres, logikai, egész, felsorolt és lebegőpontos konstansokat használhatunk. A felsorolt (**enum**) konstansok definiálásával a típusokat ismertető fejezetben részletesen foglalkozunk.

A C++ nyelv logikai konstansai az igaz értéket képviselő **true** és a hamis értékű **false**. A nyelv egyetlen mutatókonstanssal rendelkezik a nullával (0), melyet gyakran a **NULL** szimbólummal jelölünk.

2.3.1 Egész konstansok

Az egész konstansok számszámok sorozatából állnak. A számszámok decimális (10-es), oktális (8-as) vagy hexadecimális (16-os) számszámrendszerbeli jegyek lehetnek. Az egész konstansok, amennyiben nem előzi meg őket negatív (-) előjel, pozitív értékeket jelölnek.

Decimális (10-es alapú) egész számokat jelölnek azok a konstansok, amelyeknek első számszámjegye nem 0, például:

2003, -1989, 23, -1, 0

Oktális (8-as alapú) egész konstansok első jegye 0, amelyet oktális számszámjegyek követnek:

03712, -03706, 040, -01, 0

Hexadecimális (16-os alapú) egész konstansokat a 0x, illetve a 0X előtag különbözteti meg az előző két konstansfajtától. Az előtagot hexadecimális jegyek követik:

0x7cA, -0x7c6, 0x20, -0x1, 0

Mint látni fogjuk, a C++ nyelvben egész számokat különböző típusok reprezentálnak. Az egyes típusok közötti eltérés az előjel értelmezésében és a tárolási méretben jelentkezik. A konstansok megadásakor az érték után elhelyezett betűvel írhatjuk elő a annak értelmezését.

A fenti példákban közönséges egész számokat adtunk meg. Ha azonban előjel nélküli (**unsigned**) egészet kívánunk használni, akkor az **u** vagy az **U**

65535u, 0177777u, 0xFFFFu

Nagyobb előjeles egészek tárolására az ún. hosszú (**long**) egészet használjuk, amelyet a szám után helyezett **l** (kis L) vagy **L** betűvel jelölünk:

19871207L, 0x12f35e7L

További lehetőségként az U és L betűket együtt használva előjel nélküli hosszú (unsigned long) egész konstansokat is megadhatunk:

308700774UL, 0xB8000000LU

A C++ nyelv utóbbi változataiban az L betűt kétszer alkalmazva 64-bites egész konstansokat is megadhatunk, amelyek előjelesek (LL) és előjel nélküliek (ULL) egyaránt lehetnek:

28007786025507LL, 0xC0400BF78B972D1DLLU

2.3.2 Karakterkonstansok

Az ANSI (egybájtos) karakterkonstansok egyzeres idézőjelek (' - aposztróf) közé zárt egyetlen karaktert tartalmazó konstansok:

'a', '1', '0', 'é', 'ab', '01'

Az egyetlen karaktert tartalmazó karakterkonstansok által képviselt számérték a karakter 8-bites ANSI kódja. A két karaktert tartalmazó *unicode* karakterkonstansok számértéke 16-bites:

L'A', L'ab',

Bizonyos szabványos vezérlő- és speciális karakterek megadására az ún. *escape* szekvenciákat használhatjuk. Az *escape* szekvenciában a fordított osztásjel (*backslash* - \) karaktert speciális karakterek, illetve számok követik, mint ahogy az a következő táblázatból is látható.

Értelmezés	ASCII karakter	Escape szekvencia
csengető	BEL	'\a'
visszatörles	BS	'\b'
lapedobás	FF	'\f'
újsor	NL (LF)	'\n'
kocsi-vissza	CR	'\r'
vízszintes tabulálás	HT	'\t'
függőleges tabulálás	VT	'\v'
aposztróf	'	'\''
idézőjel	"	'\"'
<i>backslash</i>	\	'\\'
kérdőjel	?	'\?'
ANSI karakter oktális kóddal	ooo	'\ooo'
ANSI karakter hexa. kóddal	hh	'\xhh'

2.3.3 Lebegőpontos konstansok

A lebegőpontos konstans olyan decimális szám, amely előjeles, valós számot reprezentál. A valós szám általában egész részből, tizedes tört részből és kitevőből áll. Az egész- és tört részt tizedes pont (.) kapcsolja össze, míg a kitevő (10 hatványkitevője) az *e*, vagy az *E* betűt követi:

```
1, -2., 100.45, 2e-3, 11E2, -3.1415925, 31415925E-7
```

A C++ nyelvben a lebegőpontos értékek a tárolásukhoz szükséges memóriaterület méretétől függetlenül – ami a tárolt valós szám pontosságát és nagyságrendjét egyaránt meghatározza – lehetnek egyszerűes (*float*), kétszereses (*double*) vagy nagy (*long double*) pontosságú számok. A lebegőpontos konstansok alaphelyzetben dupla pontosságú értékek. Vannak esetek, amikor megelégszünk egyszerűes pontosságú műveletekkel is, ehhez azonban a konstansokat is egyszerűes pontosságúként kell megadni a számot követő *f* vagy *F* betűk felhasználásával:

```
3.1415F, 2.7182F
```

Nagy pontosságú számítások elvégzéséhez nagy pontosságú lebegőpontos konstansokat kell definiálnunk az *l* (kis *L*) vagy az *L* betű segítségével:

```
3.1415926535897932385L, 2.7182818284590452354l
```

2.4 Sztringkonstansok (litterálok)

Az ANSI sztringlitterál, amit sztringkonstansnak is szokás hívni, kettős idézőjelek közé zárt karaktersorozatot jelent:

```
"Ez egy ANSI sztringkonstans!"
```

A megadott karaktersorozatot a statikus memóriaterületen helyezi el a fordító, és ugyancsak eltárolja a sztringet záró '\0' karaktert (nullás bájtot) is. A sztringkonstans tartalmazhat *escape* szekvenciókat is, melyek esetén csak a nekik megfelelő karakter (egy bájti) kerül tárolásra:

```
"\nEz az első sor!\nA második sor!\n"
```

Egymas után elhelyezkedő sztringkonstansokat egyetlen sztringlitterálként tárolja a fordító:

```
"Hosszú szöveget két vagy " " több darabra tördelhetünk."
```

A széles karaktereket tartalmazó unicode sztringkonstansok előtt az *L* betűt kell használnunk:

```
L"Ez egy unicode sztringkonstans!"
```

<i>Írásjel</i>	[]	Tömb kijelölése, méretének megadása,
	()	A paraméter- és az argumentumlista kijelölése,
	{ }	Kódblokk vagy függvény behatárolása,
	*	A mutatótípus jelölése a deklaráción belül,
	,	A függvény-argumentumok elválasztása,
	:	Címke elválasztása,
	?	Az utasítás végének jelölése,
	..	Változó hosszúságú argumentumlista jelölése,
	#	Előfordító direktíva jelölése.

Az írásjel szerepe

Az írásjelek a C++ nyelvben olyan szimbólumokat jelölnek, amelyeknek csak szintaktikai szerepük van. Az írásjeleket általában azonosítók elkülönítésére, a programokód egyes részeinek kijelölésére használjuk, és semmilyen műveletet sem definiálnak. Néhány írásjel egyben operátor is.

static_cast	i	i =	%	% =	&	&&	&=	()	*	*
const_cast	+	++	+=	,	-	--	--	->	.	/
dynamic_cast	/=	<	<=	<<	>	>=	>	>	>=	>>
reinterpret_cast	>>=	?:	[]	✓	✓=		=		~	::
sizeof	.	*	->*	new	delete					

Az operátorok olyan (egy vagy több karakterből álló) szimbólumok, amelyek megmondják, hogyan kell feldolgozni az operandusokat. Az operátorok részletes ismertetésére a további fejezetekben kerül sor. Itt csak azért teszünk róluk említést, mivel szintén a C++ nyelv alapegységei (tokenjei). A következő táblázatban minden magyarázat nélkül felsoroljuk a C++ nyelv (szabványos) operátorait:

2.6 Operátorok és írásjelek

```
/* Az alábbi részben megadjuk
a változók definícióját */
int i=0; /* segédváltozó */

// Az alábbi részben megadjuk
// a változók definícióját
int i=0; // segédváltozó
```

nem kell lezárni, hatása a sor végéig terjed. leken kívüli a // (két perjel) is használható. A // jel használata esetén a megjegyzést egyen. A C++ nyelvben a megjegyzések programban történő elhelyezésére a /* ... */ je-ram forráskódja jól dokumentált, ezáltal egyszerűen értelmezhető, jól olvasható le- A megjegyzések olyan karaktersorozatok, melyek elhelyezésének célja, hogy a prog-

2.5 Megjegyzések a programban

3. A C++ program szerkezete

A C++ nyelven megírt program egy vagy több forrásfájlban (fordítási egységben, modulban) helyezkedik el, melyek kiterjesztése általában .CPP A programhoz általában ún. deklarációs (*include*, *header*, *fej*-) állományok is csatlakoznak, melyeket az *#include* előfordító utasítás segítségével építünk be a forrásállományokba.

3.1 Egyetlen modulból felépülő C++ program

A C++ program fordításához szükséges deklarációkat a *main()* függvénynel azonos forrásfájlban, de tetszőleges számú más forrásállományban is elhelyezhetjük. A *main()* függvény kitüntetett szerepe abban áll, hogy kijelöli a program belépési pontját, vagyis a program futása a *main()* függvény indításával kezdődik. Ezért érthető az a megkövetés, hogy minden C++ programnak tartalmaznia kell egy *main()* nevű függvényt, de csak egy példányban. Az alábbi példában egyetlen forrásfájlból álló C++ program szerkezete követhető nyomon:

```
#include <iostream>
#define EGY 1
#define KETTO 2

using namespace std;
int sum(int, int); // globális definíciók és
// deklarációk

int e=8;

// A main függvény definíciója
void main()
{
    int a; // Lokális definíciók és
    a = EGY; // deklarációk, utasítások
    int b = KETTO;
    e=sum(a,b);
    cout<<"Az összeg: "<<e<<endl;
    cin.get();
}

// A sum függvény definíciója
int sum(int x, int y) {
    int z; // Lokális definíciók és
    z=x+y; // deklarációk, utasítások
    return z;
}
```

A fenti példaprogram két függvény tartalmaz. A *sum()* függvény a paraméterként kapott két számot összeadja, és függvényértékként ezt az összeget szolgáltatja.

Tároljuk a példaprogramot a `CppProg.cpp` állományban! A futtatható fájl előállítás (*translation*) két fő lépésben megy végbe:

- Az első lépés a `CppProg.cpp` forrásfájl fordítása (*compiling*), melynek során olyan közbelső állomány jön létre, amely a program adatait és gépi szintű utasításait tartalmazza. (IBM PC számítógépen ezt a fájlt tárgymodulnak (*object modul*) hívjuk, és általában .OBJ (.O) kiterjesztésű állományban helyezkedik el. A fordítás eredménye még nem futtatható, hiszen tartalmazhat olyan (például könyvtári) hivatkozásokat, amelyek még nem kerültek feloldásra.
- A második lépés feladata a futtatható állomány összeállítása (*linking*). Itt fontos szerepet játszanak a C++ szabványos függvények kódját tartalmazó könyvtárak, melyek általában .LIB (.A) kiterjesztésű állományokban helyezkednek el. (IBM PC számítógépen a keletkező futtatható programot .EXE fájl tartalmazza.)

3.2 Több modulból álló C++ program

A C++ nyelv tartalmaz eszközőket a moduliális programozás elvének megvalósításához. A moduliális programozás lényege, hogy minden modul önálló fordítási egységet képez, melyeknél érvenyesül az adatrejtés elve. Mivel a modulok külön-külön lefordíthatók, nagy program fejlesztése, javítása esetén nem szükséges minden modult újrafordítani. Ez a megoldás jelentősen csökkenti a futtatható program előállításának idejét.

Az adatrejtés elvét a későbbiekben tárgyalásra kerülő fájlszintű érvenyeségi tartomány (láthatóság, *scope*), névtérületek (*namespace*) és a tárolási osztályok biztonságja. Ezek megfelelő használatával a modul bizonyos nevei kívülről (*extern*) is láthatók lesznek, míg a többi név elérhetősége a modutra korlátozódik.

A több modulból álló C++ program fordításának bemutatásához az előző alfejezet példáját vágjuk ketté! A `CppProg1.cpp` fájl csak a *main()* függvényt és a `CppProg2.cpp` állományban elhelyezkedő *sum()* függvény leírását (prototípusát) tartalmazza. Az *extern* kulcsszó jelzi, hogy a *sum()* függvényt más modulban kell a szerkesztőnek keresnie.

```
#include <iostream>
#define EGY 1
#define KETWO 2

// Előfordító utasítások

using namespace std;
// Globális definíciók és
extern int sum(int, int); // deklarációk
int e=8;
```



```
// A main függvény definíciója
void main()
{
    int a;
    // Lokális definíciók és
    // deklarációk, utasítások
    int b = KETTŐ;
    e=sum(a,b);
    cout<<"Az összeg: "<<e<<endl;
    cin.get();
}
```

A *CppProg2.cpp* fájlban csak a *sum()* függvény található:

```
// A sum függvény definíciója
int sum(int x, int y) {
    int z;
    // Lokális definíciók és
    // deklarációk, utasítások
    return z;
}
```

A futtatható fájl előállításakor minden modul külön le kell fordítanunk, *(CppProg1.obj, CppProg1.obj)*. A keletkező tárgymodulokat a szerkesztő (*linker*) építi össze a megfelelő könyvtári hivatkozások feloldásával. A több modul tartalmazó C++ programok készítését a különböző fejlesztőeszközök a projekt (*project*) koncepció bevezetésével támogatják. A projektbe foglalt forrásállományok fordítása és szerkesztése automatikusan végbemegy. A keletkező futtatható fájl általában a projekt névet viseli.

Általánosan is elmondható, hogy a C++ forrásprogram előfordító utasítások, deklarációk és definíciók, utasításblokkok és függvények együttese. Az egyes részek további tagolását és ismertetését a következő fejezetekben végezzük.

4. Alaptípusok, változók és konstansok

A C++ nyelvben minden felhasználható névről meg kell mondannunk, hogy mi az, és hogy mire szeretnénk használni. E nélkül a fordító általában nem tud mit kezdeni az adott névvel. A C++ nyelv szóhasználatával élve mindig deklarálnunk kell az általunk alkal-

A deklaráció (leírás) során csak a név tulajdonságait (típus, tárolási osztály, láthatóság stb.) közöljük a fordítóval. Ha azonban azt szeretnénk, hogy az adott deklarációnak megfelelő memóriafoglalás is végbemenjen, definíciót kell használnunk. A definíció tehát olyan deklaráció, amely helyfoglalással jár. Ugyanazt azt a nevet többször is deklarálhatjuk, azonban az egymást követő deklarációknak egyezniük kell. Ezzel szemben valamely név definíciója csak egyetleneszer szerepelhet a programban. A fordító számára a deklaráció (definíció) során közölt egyik legfontosabb információ a típus.

4.1 A C++ nyelv típusai

A C++ nyelv típusait többféleképpen csoportosíthatjuk. Az első csoportosítást a tárolt adatok jellege alapján végezzük:

Csoport	Mely típusokat tartalmazza
Integral (egész jellegű) típusok, (melyek felhasználhatók a switch utasításban.)	bool, char, wchar_t, int, enum
Aritmetikai típusok (elvégezhető rajtuk a négy alapművelet).	Az integral típusok, kiegészítve a float, double és a long double típusokkal.
Skálár típusok (for ciklusváltozókhoz használhatók).	Az aritmetikai adattípusok, a referenciák és a mutatók
Aggregate (több érték tárolására alkalmas típusok).	Tömb és a felhasználói típusok (class, struct, union).

Az előzőeknél sokkal egyszerűbb, azonban bizonyos szempontból kibővített értelmezését jelenti a típusoknak az a csoportosítás, amely az alaptípusokat (base types) és a származtatott (derived) típusokat különbözteti meg. Az alaptípusokhoz a char, az előjeles és előjel nélküli egészek és a lebegőpontos típusok tartoznak (ez nem más, mint az aritmetikai típusok csoportja az enum típus nélkül). Az elemi adattípusok jellemzőit (a következő oldalon) táblázatban foglaltuk össze.

A származtatott típusok csoportja az alaptípusok felhasználásával felépített tömb-, függvény-, mutató-, osztály-, struktúra- és uniótípusokat tartalmazza.

Adattípus	Értékkészlet	Méret (bájt)	Pontosság (számlégy)
bool	false, true	1	
char	-128..127	1	
signed char	-128..127	1	
unsigned char	0..255	1	
wchar_t	0..65535	2	
int	-2147483648..2147483647	4	
unsigned int	0..4294967295	4	
short	-32768..32767	2	
unsigned short	0..65535	2	
long	-2147483648..2147483647	4	
unsigned long	0..4294967295	4	
long long	-9223372036854775808.. 9223372036854775807	8	
unsigned long long	0..18446744073709551615	8	
float	3.4E-38..3.8E+38	4	6
double	1.7E-308..1.7E+308	8	15
long double	3.4E-4932..3.4E+4932	10	19

A csoportosítást ki kell egészítenünk, egy olyan típusnévvel, amely éppen a típus hiányát jelzi (üres típus) – ez a név a **void**. A **void** típuselőírás használatára a későbbiek során, a mutatókat és a függvényeket tárgyaló fejezetekben, visszatérünk.

4.1.1 Típuselőírások, típusmódosítók

Az alaptípusokhoz tartozó **char**, **int** és **double** típuselőírásokhoz bizonyos más kulcsszavakat (típusmódosítókat) kapcsolva, újabb típuselőírásokhoz jutunk, amelyek értelmezése elter a kindulási előírástól. A típusmódosítók hatásukat kétféle módon fejtik ki. A **short** és a **long** módosítók a tárolási hosszát, míg a **signed** és az **unsigned** az előjel értelmezését szabályozzák.

A **short int** egy rövidebb (2 bájt) tárolt), a **long int** egy hosszabb (4-bájtos), míg a **long long int** egy 64-bites egész típust definiál. A **long double** típus az adott számtípuson értelmezett legnagyobb pontosságú lebegőpontos típust jelöli.

Az egész típusok lehetnek előjelesek (*signed*) és előjel nélküliek (*unsigned*). Az egész típusok (**int**, **short int**, **long int**, **long long int**) alapértelmezés szerint pozitív és negatív egészek tárolására egyaránt alkalmasak, míg a **char** típus előjelenek értel-

mezése implementációfüggő. A fenti öt típus előjeles vagy előjel nélküli volta egyértelművé tehető a **signed**, illetve az **unsigned** típusmódosítók megadásával.

A típusmódosítók önmagukban típuselőírásaként is használhatók. Az alábbiakban (ábcé-sorrendben) összefoglaltuk a lehetséges típuselőírásokat. Az előírások soronként azonos típusokat jelölnek.

```
bool
char
wchar_t
double
enum típusnev
float
int, signed, signed int
long double
long int, long, signed long, signed long int
long long int, long long, signed long long, signed long long
short int, short, signed short, signed short int
struct típusnev
class típusnev
union típusnev
unsigned char
unsigned int, unsigned
unsigned long, unsigned long int
unsigned long long, unsigned long long int
unsigned short, unsigned short int
void
```

4.1.2 Típusmódosítók

A típuselőírásokat típusmódosítóval együtt használva a deklarált azonosítóhoz az alábbi két tulajdonság egyikét rendelhetjük. A **const** kulcsszóval olyan nevet definiálhatunk, melynek értéke nem változtatható meg (csak olvasható).

A **volatile** típusmódosítóval olyan név hozható létre, melynek értékét a programunktól független kód (például egy másik futó folyamat vagy szál) is megváltoztathatja. A **volatile** közli a fordítóval, hogy nem tud minden, ami az adott változóval történhet. (Ezért például a fordító minden egyes, ilyen tulajdonságú változóra történő hivatkozásakor, a memóriából veszi fel a változóhoz tartozó értéket.)

```
int const
const int
volatile char
long int volatile
```

4.1.3 A felsorolt típus (*enum*)

Az **enum** olyan adat típust jelöl, melynek lehetséges értékei egy konstanshalmazból kerülnek ki. Az **enum** típust konstansnevek felhasználásával származtatjuk:

```
enum azonosító { felsorolás }
```

A felsorolásban szereplő konstansok az **int** típus értéktartományából vehetnek fel értéket. Nézzünk néhány példát a felsorolt típus létrehozására!

```
enum valasz { igen, nem, talan};
```

A fenti programsor feldolgozása után létrejön a felsorolt típus (**enum valasz** vagy **valasz**), és három egész konstans *igen*, *nem* és *talan*. A fordító a felsorolt konstansoknak balról jobbra haladva, nullával kezdve egész értékeket feleltet meg. A példában az *igen*, *nem* és *talan* konstansok értéke rendre 0, 1 és 2.

Ha a felsorolásban a konstans nevét egyelősség jel és egy egész érték követi, akkor a konstanshoz a fordító a megadott számot rendeli, és a tőle jobbra eső konstansok ezen kiindulási értéktől kezdődően kapnak értéket:

```
enum valasz { igen, nem=10, talan};
```

Ekkor az *igen*, *nem* és *talan* konstansok értéke rendre 0, 10 és 11 lesz. A felsorolásban azonos értékek többször is szerepelhetnek:

```
enum valasz { igen, nem=10, lehet, talan=10};
```

A **enum** típust elsősorban csoportos konstansdefiniálásra használjuk – ekkor a típusnevet el is hagyhatjuk:

```
enum { also=-2, felso, kiraly=1, ass };
```

ahol a konstansok értékei sorban -2, -1, 1 és 2.

Az alábbi deklarációt használva, a programrésztlet C++-ban figyelmeztetéshez vezet:

```
enum szin { feketek, kek, zold};

int kod;
enum szin kol;

kol = zold;
kod = kol;
kol = 23;
kol = szin(23); // ok!
// figyelmeztetés!
// rendben, a kod értéke 2 lesz
// rendben
```

Az **enum** deklarációban megadott név típusnévként is használható a kulcsszó megadá-
sa nélkül:

```
enum Boolean {False, True};

Boolean x = False;
enum Boolean y = x;
```

4.2 Egyszerű változók definíciója

A C++ program memóriában létrehozott tárolóit nével látjuk el, hogy tudjunk rájuk hivatkozni. A név segítségével a tárolóhoz értéket rendelhetünk, illetve lekérdezhetjük az eltárolt értéket. A nével ellátott tárolókat a programozási nyelvekben változónak szokás nevezni. Nézzük először általános formában, hogyan néz ki a változók defini-
ciója (deklarációja):

$\langle \text{tárolási osztály} \rangle \text{ típus } \langle \text{típus} \dots \rangle \text{ változónev} (= \text{kezdőérték}) \langle , \dots \rangle ;$

Az általánosított formában a $\langle \rangle$ jelek az opcionálisan megadható részeket jelölik, míg a három pont az előző definíciós elem ismételtelhetőségére utal. Külön felhívjuk a fi-
gyelmet arra, hogy a deklarációs sort pontosvesszővel kell lezárni.

```
int alpha;
int beta=4;
int gamma, delta=9;
```

A deklarációban a típust megelőzheti néhány alapszó (**auto**, **register**, **static** vagy **extern**), amelyek a változó tárolásával kapcsolatosan tartalmaznak előírásokat. Az elő-
írások, amiket tárolási osztálynak nevezünk, meghatározzák a változó elhelyezkedését, láthatóságát és élettartamát. Minden változóhoz tartozik tárolási osztály, még akkor is, ha azt külön nem adtuk meg. Ha a változót a függvényeken kívüli definíciójuk, akkor az alapértelmezés szerint globális (más modulból elérhető - **extern**) tárolási osztállyal rendelkezik, míg a függvényeken belüli definíált változók alaphelyzetben automatikus (**auto**) változók.

Az **extern** tárolási osztályú változók akkor is kapnak kezdőértéket (nullát), ha nem adunk meg semmit. A függvényen belüli definíalt automatikus változókat tetszőleges kifejezéssel inicializálhatjuk.

```
int av = sin(log(rand()));
```

C++-ban (ellentétben a C nyelvvél) a globális elérési változók kezdőértékeként szin-
tén tetszőleges kifejezést megadhatunk:

```
long ev = (1 << 4) + exp(-1);
```

4.3 Saját típusok előállítása

A C++ nyelv típusnevei, a típusmódosítók és a típusminősítők megadásával, általában több alapszóból tevődnek össze, például:

```
volatile unsigned long int
```

Definiáljunk a fenti típus felhasználásával egy kezdőérték nélküli változót!

```
volatile unsigned long int idozites;
```

A C++ nyelv tartalmaz egy speciális tárolási osztályt (**typedef**), amely lehetővé teszi, hogy érthetőbb típusokhoz szinonim neveket rendeljünk:

```
typedef volatile unsigned long int tido;
```

Az új típussal már sokkal egyszerűbb az *idozites* változót definiálni:

```
tido idozites;
```

Különösen hasznos a **typedef** használata összetett típusok esetén, ahol a típusdefiniáció felírása nem mindig triviális. A típusok készítése azonban mindig eredményes lesz, ha a következő tapasztalati szabály betartjuk:

- Írjunk fel egy kezdőérték nélküli változódefiniációt, ahol az a típus szerepel, amelyhez szinonim nevet kívánunk kapcsolni!
- Írjuk a definiáció elé a **typedef** kulcsszót, ami által a megadott név nem változót, hanem típust fog jelölni!

Tekintsünk néhány gyakran használt típusnevet!

```
typedef unsigned char byte;
typedef unsigned short word;
typedef long long int64;
```

4.4 Konstansok a C++ nyelvben

A C++ nyelvben többféle módon használhatunk konstansokat. Az első lehetőség a **const** típusminősítő megadását jelenti a változódefiniációban. A változók értéke általában megváltoztatható:

```
int a;
a=7;
```

Ha azonban a definiációban szerepel a **const** kulcsszó, a változó „csak olvasható” lesz, vagyis értékét nem lehet közvetlenül megváltoztatni. (Ekkor a definiációban kötelező a kezdőérték megadása.)


```
const int a=30;
```

```
a=7;
```

```
// Hibafigyelést kapunk a fordítótól.
```

A másik, szintén gyakran használt megoldás, amikor az előfordító *#define* utasításával létrehozott makrók hordoznak konstans értékeket. Az előfordító által használt neveket csupa nagybetűvel szokás írni:

```
#define FEKETE 0
#define KEK 1
#define ZOLD 2
#define PIROS 4
#define PI 3.14159265
```

```
void main()
```

```
{
```

```
int a=KEK;
```

```
double f;
```

```
a += PIROS;
```

```
f = 90*PI/180;
```

```
}
```

Ezek a szimbolikus nevek valójában konstans értékeket képviselnek. Az előző programrészt az előfordítás után:

```
void main()
```

```
{
```

```
int a=1;
```

```
double f;
```

```
a+=4;
```

```
f=90*3.14159265/180;
```

```
}
```

A harmadik lehetőség, az **enum** típus használatát jelenti, ami azonban csak egész (**int**) típusú konstansok esetén alkalmazható. Az előző példában szereplő színekonstansokat az alábbi alakban is előállíthatjuk:

```
enum szinek {fekeke, kek, zold, piros=4};
```

```
void main()
```

```
{
```

```
int a = kek;
```

```
a += piros;
```

```
}
```

Az **enum** és a **const** konstansok igazi konstansok, hisz nem tárolja őket a memóriában a fordító. Míg a *#define* konstansok a definiálás helyétől a fájl végéig fejtik hatásukat, addig az **enum** és a **const** konstansokra a szokásos C++ láthatósági és élettartam szabályok érvényesek. Altalában is elmondható, hogy a C++ nyelvben javasolt elkerülni a *#define* konstansok használatát, mivel bizonyos esetekben hibát vihettek a forrásprogramba.

4.5 Ertékek, címek, mutatók és referenciák

A változók általában az értékadás során kapnak értéket, melynek általános alakja:

változó = *érték*;

A C++ nyelvén az értékadó operátor tartalmazó utasítás valójában egy kifejezés, amit a fordítóprogram kiértékel. Az értékadás operátorának bal- és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban lényegileg különbözőznek egymástól. A bal oldalon szereplő kifejezés azt a változót jelöli ki (címszi meg) a memóriában, ahova a jobb oldalon megadott kifejezés értéket be kell tölteni.

4.5.1 Balérték és jobbérték

A fenti alakból kiindulva a C++ nyelv külön nevet ad a kétfajta kifejezésnek. Annak a kifejezésnek az értéke, amely az egyenlőségjel bal oldalán áll, a balérték (*lvalue*), míg a jobb oldalon szereplő kifejezés értéke a jobbérték (*rvalue*). Vegyünk példaként két egyszerű értékadást!

```
int a;
a = 12;
a = a + 1;
```

Az első értékadás során az *a* változó mint balérték szerepel, vagyis a változó címe jöböl ki azt a tárolót, ahova a jobb oldalon megadott konstans értéket be kell másolni. A második értékadás során az *a* változó az értékadás mindkét oldalán szerepel. A bal oldalon álló *a* ugyancsak a tárolót jelöli ki a memóriában (*lvalue*), míg a jobb oldalon álló *a* egy jobbérték kifejezésben szerepel, melynek értékét (13) a fűtő program határozza meg az értékadás elvégzése előtt.

4.5.2 Ismerkedés a mutatóval és a referenciával

A *mutatók* használata a C++ nyelvben alapvető követelmény. Ebben a részben csak a mutatók fogalmát vezetjük be, míg alkalmazásukkal a könyvünk további fejezeteiben részletesen foglalkozunk.

Definiáljunk egy egész típusú változót!

```
int x = 23;
```

A definíció hatására a memóriában létrejön egy (*int* típusú) tároló, amelybe bemásolódik a kezdőérték.

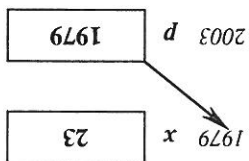
Az

```
int * p;
```

definiáció hatására szintén létrejön egy tároló, melynek típusa `int*`. Ez a változó `int` típusú változók címének tárolására használható, melyet a „címe” művelet során szerezhünk meg, például:

```
p = &x;
```

A művelet után az `x` név és a `*p` érték ugyanarra a memóriaterületre hivatkoznak. (A `*p` kifejezés a „`p` által mutatott” tárolót jelöli.)



Ennek következtében a

```
*p = x + 12;
```

kifejezés feldolgozása során az `x` változó értéke 35-re módosul.

A *hivatkozási* (referencia) típus felhasználásával már létező változókra hivatkozhatunk, alternatív névet definiálva. A definíció általános formája:

```
típus &azonosító = változó;
```

A referencia definiálásakor kötelező kezdőértéket adnunk. A fentiekben definiált `x` változóhoz referenciát is készíthetünk:

```
int x = 23;
int &r = x;
```

Ellentétben a mutatókkal, a referencia tárolására általában nem jön létre külön változó. A fordító egyszerűen második névként egy új nevet ad az `x` változónak (`r`).

```
r, x
23
```

Ennek következtében az alábbi kifejezés kiértékelése után szintén 35 lesz az `x` változó értéke:

```
x = x + 12;
```

Ellentétben a `p` mutatóval, melynek értéke (ezáltal a mutatott tároló) bármikor megváltoztatható, az `r` referencia a változóhoz kötött.

Ha egy referenciát *konstans* értékkel, vagy *előző típusú* változóval inicializálunk fordítási hibát kapunk. Az ilyen eseteket csak akkor kezelni a fordító, ha ún. konstans referenciát készítünk. Ekkor a fordító először létrehozza a hivatkozás típusával megegyező típusú tárolót, majd pedig inicializálja az egyenlőséggel után szereplő jobbérték kifejezés értékével.

```
const int & n = '\n';
unsigned b = 2003;
const int & r = b;
```

Saját mutató-, illetve referenciátípus szintén létrehozható a **typedef** tárolási osztály felhasználásával:

```
int x = 23;
typedef int & tr1;
typedef int * tp1;
```

```
tr1 r = x;
tp1 p = &x;
```

Mutatóhoz referenciát más típusú változókhoz hasonlóan készíthetünk. Segítségével például, kiküszöbölhetjük a kétszeres indirektitású mutatók függvényparaméterként való alkalmazását.

```
int n = 123;
int * p = &n;
int & rp = p;
// rp referencia a p mutatóhoz
*rp = 7;
// n értéke 7 lesz
```

Ugyanez a **typedef** segítségével:

```
int n = 123;
typedef int * tp1;
tp1 p = &n;
tp1 & rp = p;
// rp referencia a p mutatóhoz
*rp = 7;
// n értéke 7 lesz
```

Felhívjuk a figyelmet azonban arra, hogy referenciához mutatót nem definiálhatunk - ezért az alábbi programrészt harmadik sora fordítási hibát okoz:

```
int n = 123;
int & r = n;
int & * pr = &r; // referenciára mutató pointer nem használható
// a p mutató az r referencián keresztül az n-re mutat
```

4.5.2.1 A `void` * típusú általános mutatók

Az előző példákban a pointerekkel `int` típusú változókra mutattunk. A változó eléréséhez azonban nem elegendő annak címét tárolnunk (ezt tartalmazza a mutató), hanem definiálnunk kell a tároló méretét is, amit a mutató típusa közvetít a fordító felé. A C++ nyelv típus nélküli (`void`), ún. általános mutatók használatát is lehetővé teszi:

```
int x;
void * ptr = &x;
```

amelyek azonban sohasem jelölnek ki tárolót.

Amenyiben ilyen mutatóval szeretnénk a hivatkozott változónak értéket adni, akkor felhasználói típuskonverzióval (*type-casting*) típust kell rendelnünk a cím mellé. A típus-átalakítást többféleképpen is elvégezhetjük:

```
int x;
void * ptr = &x;
*(int *)ptr = 123;
typedef int * iptr;
*iptr(ptr) = 123;
static_cast<int*>(ptr) = 123;
```

Mindhárom, indirekt módon elvégzett értékadást követően, az `x` változó értéke `123` lesz.

4.5.2.2 Többszörös indirektosságú mutatók

A mutatókat többszörös indirektosságú kapcsolatok esetén is használhatunk. Ekkor a mutatók definíciójában több csillag (*) szerepel. Tekintsünk néhány definíciót, és mondjuk meg, hogy mi a létrehozott változó!

```
int x;
x egy egész típusú változó.

int *p;
p egy int típusú mutató (amely int változóra mutathat).

int **q;
q egy int* típusú mutató (amely int* változóra, vagyis egészre mutathat).
```

Már említettük, hogy a C++ nyelven megadott típusok eléggé összetettek is lehetnek. A bonyolultság feloldásában, a deklarációk értelmezésében a mutatók esetén is ajánlott a `typedef` használata:

```
typedef int * iptr; // egészsre mutató pointer típusa
iptr p, *q;
```

vagy

```
// egésze mutató pointer típusa
typedef int * iptr;

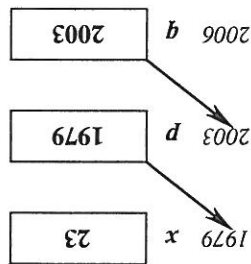
// iptr típusú változóra mutató pointer típusa
typedef iptr * ipptr;

iptr p;
ipptr q;
```

A definíciók megadása után kijelenthetjük, hogy az

```
x = 23;
p = &x;
q = &p;
x = x + *p + **q;
```

utasítások lefutását követően az x változó értéke 69 lesz. Az elmondottakat jól szemlélíti az alábbi ábra:



5. Operátorok és kifejezések

Az eddigiek során gyakran használtunk olyan utasításokat (mint például az értékadás), amelyek pontosvesszővel lezárt kifejezésből álltak. A kifejezések egyetlen operandusból, vagy az operandusok és a műveleti jelek (operátorok) kombinációjából épülnek fel. A kifejezés kiértékelése valamilyen érték kiszámításához vezethet, függvényhívást idézhet elő, vagy mellékhatást (*side effect*) okozhat. Az esetek többségében a fenti három tevékenység valamilyen kombinációja megy végbe a kifejezések feldolgozása (kiértékelése) során.

Az operandusok a C++ nyelv azon elemei, amelyeken az operátorok kifejtik a hatásukat. Azokat az operandusokat, amelyek nem igényelnek további kiértékelést elsődleges (*primary*) kifejezéseknek nevezzük. Ilyenek az azonosítók, a konstans értékek, a stringlitterálok és a zárójelben megadott kifejezések. Hagyományosan az azonosítókhoz soroljuk a függvényhívásokat, valamint a tömbelem- és a struktúratag-hivatkozásokat is.

A kifejezések kiértékelése során az operátorok lényeges szerepet játszanak. Az operátorokat több szempont alapján lehet csoportosítani. A csoportosítást elvégezhetjük az operandusok száma szerint. Az egyoperandusú (*unary*) operátorok esetén a kifejezés

általános alakja:

op operandus vagy operandus op

Az első esetben, amikor az operátor (*op*) megelőzi az operandust előrevetett (*prefixes*), míg a második esetben hátravetett (*postfixes*) alakról beszélünk:

-a ++a sizeof (a) float (a) &a

Az operátorok többsége két operandussal rendelkezik – ezek a kétoperandusú (*binary*) operátorok:

operandus1 op operandus2

Ebben a csoportban a hagyományos aritmetikai műveletek mellett megtalálhatók a bitműveletek elvégzésére szolgáló operátorok is:

a+b a!=b a<<2 a+=b a & 0xFF00

A C++ nyelvben egyetlen háromoperandusú operátor, a feltételes operátor használható:

a < 0 ? -a : a

Az alábbi táblázatban a C++ nyelv műveleteit precedencia szerint csoportosítottuk. Az első csoportban a legmagasabb precedenciájú műveletek találhatók. (A *kif* rövidítéssel a kifejezéseket jelöltük.)

1.	<i>asszociativitás: balról-jobbra</i>	<i>osztálynév::tag</i> <i>névterületnév::tag</i> <i>::név</i> <i>::minősített_név</i>
2.	<i>asszociativitás: balról-jobbra</i> függvényhívás tömbindexelés közvetett tagkiválasztás közvetlen tagkiválasztás (postfix) léptetés előre (postfix) léptetés vissza érték létrehozása futasídejű ellenőrzött típusátalakítás fordításiidejű ellenőrzött típusátalakítás ellenőrizetlen típusátalakítás konstans típusátalakítás futasídejű típusazonosítás	<i>kif(kif-lista)</i> <i>mutató[kif]</i> <i>mutató->tag</i> <i>objektum.tag</i> <i>balérték++</i> <i>balérték--</i> <i>típus(kif-lista)</i> <i>dynamic_cast<típus>(kif)</i> <i>static_cast<típus>(kif)</i> <i>reinterpret_cast<típus>(kif)</i> <i>const_cast<típus>(kif)</i> <i>typeid(kif)</i> <i>typeid(típus)</i>
3.	<i>asszociativitás: jobbról-balra</i> logikai tagadás (NEM) bitenkénti negálás + előjel - előjel (prefix) léptetés előre (prefix) léptetés vissza a címe operátor az indirektiség operátor típusátalakítás objektum bájtban kifejezett mérete típus bájtban kifejezett mérete tárterület foglалása tárterület foglалása tömb számára tárterület foglалása és inicializálása tárterület elhelyezése tárterület elhelyezése és inicializálása tárterület felszabadítása tárterület felszabadítása delete [] <i>mutató</i>	<i>! kif</i> <i>~ kif</i> <i>+ kif</i> <i>- kif</i> <i>++ balérték</i> <i>-- balérték</i> <i>&balérték</i> <i>*kif</i> <i>(típus) kif</i> <i>sizeof kif</i> <i>sizeof (típus)</i> <i>new típus</i> <i>new típus[kif]</i> <i>new típus(kif-lista)</i> <i>new (kif-lista) típus</i> <i>delete mutató</i> <i>delete [] mutató</i>

4.	objektum.* tagmutató mutató->* tagmutató	asszociativitás: balról-jobbra osztálytagra történő indirekt hivatkozás mutatóval megadott objektum tagjára való indi- rekt hivatkozás
5.	kif * kif kif / kif kif % kif	asszociativitás: balról-jobbra szorzás osztás maradék
6.	kif + kif kif - kif	asszociativitás: balról-jobbra összeadás kivonás
7.	kif << kif kif >> kif	asszociativitás: balról-jobbra biteltolás balra biteltolás jobbra
8.	kif < kif kif <= kif kif > kif kif >= kif	asszociativitás: balról-jobbra kisebb kisebb vagy egyenlő nagyobb nagyobb vagy egyenlő
9.	kif == kif kif != kif	asszociativitás: balról-jobbra egyenlő nem egyenlő
10.-14.	kif & kif kif kif kif ^ kif kif && kif kif kif	asszociativitás: balról-jobbra bitenkénti ES bitenkénti VAGY bitenkénti kizáró VAGY logikai ES logikai VAGY
15.	kif ? kif : kif	asszociativitás: jobbról-balra feltételes kifejezés balról-jobbra
16.	balérték = kif balérték * = kif balérték / = kif balérték % = kif balérték + = kif balérték - = kif balérték < <= kif balérték > >= kif balérték & = kif balérték ^ = kif balérték = kif balérték != kif	asszociativitás: jobbról-balra egyszerű értékadás szorzat megfeleltetése hányados megfeleltetése maradék megfeleltetése összeg megfeleltetése különbség megfeleltetése balra eltolít bitek megfeleltetése jobbra eltolít bitek megfeleltetése bitenkénti ES megfeleltetése bitenkénti kizáró VAGY megfeleltetése bitenkénti VAGY megfeleltetése

(3 + 4) * 5

kifejezés értéke 23. Ha az összeget zárójelbe tesszük, megváltoztatva ezzel a kiértéke-
lés sorrendjét

3 + 4 * 5

lelőben például az
szór, amit az alacsonyabb precedenciájú műveletek végrehajtása követ. Ennek megfe-
sabb precedenciával rendelkező operátort tartalmazó rész kifejezés értékelődik ki elő-
kifejezésben különböző precedenciájú műveletek találhatók. Ekkor mindig a maga-
Az operátorok precedenciája akkor játszik szerepet a kifejezés kiértékelése során, ha a

5.1.1 Az elsőbbségi (precedencia) szabály

Annak érdekében, hogy bonyolultabb kifejezéseket is helyesen tudjunk használni, meg-
kell ismerkednünk az elsőbbségi (precedencia) szabályokkal, amelyek meghatározzák
a kifejezésekben szereplő műveletek kiértékelési sorrendjét. Az egyes operátorok kö-
zötti elsőbbségi kapcsolat a műveletek táblázatban foglaltuk össze. A táblázat cso-
portjai az azonos precedenciával rendelkező operátorokat tartalmazzák. A csoportok
mellett külön jelezük az azonos precedenciájú operátorokat tartalmazó kifejezésben a
kiértékelés irányát, amit asszociativitásnak (csoportosításnak) hívunk. A táblázat első
csoportja a legnagyobb precedenciával rendelkező műveleteket tartalmazza.

5.1 Precedencia és asszociativitás

Az átdeterminálás során az egyes operátorok új értelmezést kaphatnak, azonban a fenti
táblázat szerinti precedencia és asszociativitás nem változtatható meg.

. * . : : # ? : # ##

A felhasználtói típushoz kapcsolódóan a C++ lehetőséget kínál az operátorok többsé-
gének átdeterminálására, túllerhelésére (operator overloading) az alábbi operátorok
kivételével:

17.	throw kif	asszociativitás: balról-jobbra kivétel kiváltása
18.	kif , kif	asszociativitás: balról-jobbra művelet sor
Előforduló műveleti jelek		
#		
##		
defined(azonosító)		
máskör létezésének vizsgálata		
illesztés sztringbe		
illesztés szövegbe		

35 lesz az eredmény. Ha azonban a kiindulási kifejezésben a zárójelbe a szorzatot helyezzük

$$3 + (4 * 5)$$

akkor ismét 23-at kapunk eredményként, hisz ebben az esetben a zárójelzés csak megerősítette, de nem változtatta meg a számítás menetét.

Az egyes operátorok működésének mélyebb ismerete nélkül nézzünk néhány példát a precedencia érvényesülésére! A precedencia-táblázat segítségével gyorsan felírhatjuk azt az ekvivalens kifejezést, ahol zárójeltek felhasználásával kihangsúlyozzuk a kiértékelés menetét.

Kifejezés:	
$a \& b \mid \mid c$	$++ * p++$
$a = (b \mid \mid c)$	
$(a \& b) \mid \mid c$	$++ (* (p++))$

5.1.2 A csoportosítási (asszociativitás) szabály

Egy kifejezésben több azonos precedenciájú művelet is szerepelhet. Ebben az esetben az operátortáblázat csoportszámai mellett található útmutatást kell figyelembe venni a kiértékelés során, hisz a precedencia-szabály nem alkalmazható. Az asszociativitás azt mondja meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra, vagy jobbról-balra haladva kell elvégezni.

Példaul, az értékelő utasítások csoportjában a kiértékelést jobbról-balra haladva kell elvégezni, így adott a lehetőség több változó együttes inicializálására:

```
int a, b, c;
a = b = c = 23;
```

Zárójeltek használatával most is felírhatunk egy olyan kifejezést, amely ekvivalens a fentivel;

$$a = (b = (c = 23));$$

5.2 Mellékhatások és a rövidzár kiértékelés

Bizonyos műveletek – a függvényhívás, a többszörös értékelás és a léptetés (növelés, csökkentés) – feldolgozása során a kifejezés értékeknek megjelenése mellett bizonyos változók is megváltozhatnak. Ezt a jelenséget mellékhatásnak (*side effect*) hívjuk. A mellékhatások kiértékelésének sorrendjét nem határozza meg a C++ szabvány, ezért javasolt minden olyan megoldás elkerülése, ahol a kifejezés eredménye függ a mellékhatások kiértékelésének sorrendjétől, például:

```
a[i] = i++;
std::cout<<"+n<pow(2,n)<<endl;
```

A művelettáblázatból látható, hogy a logikai kifejezések kiértékelése szintén balról jobbra haladva történik. Bizonyos műveleteknél nem szükséges a teljes kifejezést kiértékelni ahhoz, hogy egyértelmű legyen a kifejezés értéke. Példaként vegyük a logikai ES (&&) operátort, amely használata esetén a bal oldali operandus 0 értéke esetén a jobb oldali operandus kiértékelése feleslegessé válik! Ezt a kiértékelési módot rövidzár (*short-circuit*) kiértékelésnek nevezzük.

Ha a rövidzár kiértékelése során a logikai operátor jobb oldalán valamilyen mellékhatás kifejezés áll,

x || y++

az eredmény nem mindig lesz az, amit várunk. A fenti példában x nem nulla értéke esetén az y léptetésére már nem kerül sor.

5.3 Az elsődleges kifejezés operátorai

Az ANSI C nyelvben az elsődleges kifejezések a literálok (konstans értékek), az azonosítók és a zárójeles kifejezések voltak. A C++ nyelv ezt a listát bővíti a **this** kulcsszóval, a hatókörfeloldó operátorral, a nevekkel és az osztálydestruktorral ~.

literálok egész-, karakter-, valós- és karaktertípusú konstansok
this (kifejezés)
:: azonosító,
::: operátorfüggvény-név,
::: minősített név,
azonosító,
operátorfüggvény-név,
konverziós függvény neve,
~osztálynév,
minősített név, minősített osztálynév::név

5.4 Aritmetikai operátorok

Az aritmetikai operátorok csoportja a szokásos négy alapműveleten túlmenően a maradékképzés operátorát (%) is tartalmazza. Az összeadás (+), a kivonás (-), a szorzás (*) és az osztás (/) művelete egész és lebegőpontos számok esetén egyaránt használható. Az osztás egész típusú operandusok esetén egész osztást jelöl:

23 / 4	a kifejezés értéke (a hányados)	5
23 % 4	az kifejezés értéke (a maradék)	3

A C++ nyelvben a logikai típus is az egész típusok közé tartozik – a **false** értéke 0, a **true** értéke pedig 1. Az utasításokban szereplő feltételek tetszőleges kifejezések lehetnek, melyek nulla vagy nem nulla értéke szolgáltatja a logikai hamis, illetve igaz eredményt. A feltételekben gyakran kell összehasonlítani bizonyos értékeket, hogy a program további működéséről döntünk. Az összehasonlítás elvégzésére az összehasonlító operátorokat használjuk, melyeket az alábbi táblázatban foglaltuk össze:

5.5 Összehasonlító és logikai operátorok

Művelet	Kifejezés	Eredmény
két mutató kivonható egymásból	$q - p$	egész
a mutatóhoz egész szám hozzáadható	$p + i$	mutató
a mutatóból egész szám kivonható	$p - i$	mutató

Megjegyezzük, hogy a + és a - operátorok egyik vagy mindkét operandusa mutató is lehet, ekkor pointer-aritmetikáról beszélünk. A megengedett pointer-aritmetikai műveletek, ahol a q és a p (nem **void*** típusú) mutatók, az i pedig egész (**int** vagy **long**):

$$a - (b - c) / d / e$$

Az átirít kifejezés nevezőjében a zárójel használata most kiküszöbölhető, hisz a sorozás helyett ismételt osztást is használhatunk. Ezt figyelembe véve a matematikai alakítól eltérünk ugyan, de jól olvasható C++ kifejezéshez jutunk:

$$a + (b - c) / (-d * e)$$

kifejezést C++ kifejezéssé! A helyes átiráshoz a műveleti jelek cseréjén túlmenően mérlegelnünk kell a precedencia-viszonyokat. Ezzel általában nincs is gond, hiszen már az általános iskolában megtanultuk, hogy a szorzás és osztás magasabb precedenciával rendelkezik az összeadásnál, illetve a kivonásnál. Tört átirásánál biztosan jól járunk el, ha a számlálót és nevezőt is zárójelbe közé helyezzük.

$$a + \frac{b - c}{-d \cdot e}$$

Aritmetikai operátorokat leggyakrabban matematikai feladatok programozása során használunk. Példaként írjuk át az

A csoportba tartozik az egyoperandusú mínusz (-) operátor is, melynek segítségével a mögötte álló operandus értéke ellentétes előjellelre változtatható (negálható).

$$(a / b) * b + (a \% b) = a$$

Egész a és b (nem 0) esetén mindig igaz az alábbi kifejezés:

Matematikai alak	C++ kifejezés	Jelentés
$a < b$	$a < b$	a kisebb, mint b
$a \leq b$	$a \leq b$	a kisebb vagy egyenlő, mint b
$a > b$	$a > b$	a nagyobb, mint b
$a \geq b$	$a \geq b$	a nagyobb vagy egyenlő, mint b
$a = b$	$a == b$	a egyenlő b -vel
$a \neq b$	$a != b$	a nem egyenlő b -vel

Bármelyik fenti kifejezés **int** típusú, és a kifejezés értéke 1, ha a vizsgált reláció igaz, illetve 0, ha nem.

Példaként írjunk fel egy olyan kifejezést, amely igaz (**true**, 1) lesz, ha az x változó értéke nem negatív!

$$x \geq 0$$

Hogyan lehet megvizsgálni azt, hogy az x értéke -5 és 5 közé esik-e? A matematikából ismert módon írjuk fel a kifejezést:

$$-5 < x < 5$$

A kifejezést különböző x értékek esetén kiértékelve mindig 1-et kapunk. Hol a hiba? Az összehasonlító operátorok kiértékelése balról jobbra haladva történik. Először kiértékelésre kerül a $-5 < x$ kifejezés, melynek értéke 0 vagy 1 lesz. A második lépésben ezt a 0-at vagy 1-et hasonlítjuk össze 5-tel, így a kisebb reláció mindig igaz lesz.

Ahhoz, hogy bonyolultabb feltételeket is össze tudjunk hasonlítani, a relációs operátorok mellett szükségünk van a logikai operátorokra is. C++-ban a logikai **ES** (&&) és a logikai **VAGY** (||) műveletek használhatók a feltételek megfogalmazása során. Az előző feltétel egyszerűen felírható a logikai **ES** művelet alkalmazásával:

$$-5 < x \ \&\& \ x < 5$$

(Zárójeleket nem szükséges használnunk, hisz az összehasonlító operátorok magasabb precedenciával rendelkeznek a logikai operátoroknál.) Újra felhívjuk a figyelmet a „rövidzárt” kiértékelésre, ami azt jelenti, hogy a balról-jobbra haladó kiértékelés azonnal leáll, ha a kifejezés logikai értéke egyértelműen eldönthető.

Vannak esetek, amikor valamely feltétel felírása helyett egyszerűbb az ellentett feltételt megfogalmazni és alkalmazni rá a logikai tagadás (**NEM**) operátorát (!). Az előző példában alkalmazható a feltétel egyenértékű az alábbi feltétellel:

$$!(-5 > x \ || \ x \geq 5)$$

A logikai tagadás során minden relációt az ellenítetes irányú relációra, az *ES* operátor pedig a *VAGY* operátorra (illetve fordítva) cseréljük.

A C++ programokban gyakran használjuk az

```
ok == 0      kifejezés helyett a      ok
ok != 0      kifejezés helyett az      ok
```

kifejezést. Nem igazán lehet egyértelműen eldönteni, hogy melyik alakot érdemesebb alkalmazni. A második megoldás jobban olvasható („nem ok”, „ok”), de bizonyos esetekben nehezebb megérteni az *ok* változó szerepét.

Mint láttuk, a relációs operátorok mellett szükségünk van a logikai operátorokra is. C++-ban a logikai *ES* (&&), a logikai *VAGY* (||) és a logikai *NEM* (!) műveletek használhatók a feltételek megfogalmazása során. A logikai operátorok működését ún. igazságtáblával írjuk le:

logikai tagadás		logikai ES művelet		logikai VAGY művelet	
<i>a</i>	<i>!a</i>	<i>a</i>	<i>b</i>	<i>a</i>	<i>b</i>
1	0	1	1	1	1
0	1	1	0	1	0
1	0	0	1	0	1
0	1	0	0	0	0
			<i>a&& b</i>		<i>a b</i>

5.6 Lépítő operátorok

A változók értékét lépítő operátorok a magas szintű nyelvekben csak ritkán fordulnak elő. C++-nyelvvel lehetőségét biztosít valamely változó értékének egyel való növelésére ++ (*increment*), illetve egyel való csökkentésére -- (*decrement*).

A lépítő operátorok az aritmetikai típusokon kívül a mutatókra is alkalmazhatók, ahol azonban nem 1 bájtal való elmozdulást, hanem a szomszédos elemre való léptetést jelentik.

Az operátorok csak balérték operandussal használhatók, azonban mind az előrevetett, mind pedig a hátravetett forma alkalmazható:

```
int a;
```

```
// prefixes alakok:
```

```
++a;
```

```
// postfixes alakok:
```

```
++a;
```

Ha az operátorokat a fent bemutatott módon használjuk, nem látszik különbség az előrevetett és hátravetett forma között, hiszen mindkét esetben a változó értéke léptetődik. Ha azonban az operátort bonyolultabb kifejezésben alkalmazzuk, akkor a prefixes alak használata esetén a léptetés a kifejezés kiértékelése előtt megy végbe, és a változó az új értékével vesz részt a kifejezés kiértékelésében:

```
int x, n=5;
```

```
x = ++n;
```

A példában szereplő kifejezés kiszámítása után mind az x , mind pedig az n változó értéke 6 lesz.

```
double x, n=5.0;
```

```
x = n++;
```

A kifejezés feldolgozása után az x változó értéke 5.0, míg az n változó értéke 6.0 lesz.

A léptető operátorok működését jobban megértjük, ha a bonyolultabb kifejezéseket részkiifejezésekre bontjuk. A

```
int a=2, b=3, c;
```

```
c = ++a + b--;
```

```
// a 3, b 2, c pedig 6 lesz
```

kifejezés az alábbi (egy, illetve többutasításos) kifejezésekkel megegyező eredményt szolgáltat:

```
++a, c=a+b, b--;
```

```
a++; c=a+b; b--;
```

Az eggyel való növelés és csökkentés hagyományos megadása

```
a = a + 1;
```

```
a = a - 1;
```

helyett mindig érdemes a megfelelő léptető operátort használni,

```
++a; vagy a++;  
--a; vagy a--;
```

amely az áttekinthetőség mellett, gyorsabb kód létrehozását is eredményezi.

5.7 Bitműveletek

A C++ nyelv hat operátort tartalmaz, amelyekkel különböző bitenkénti műveleteket végezhetünk **char**, **short**, **int** és **long** típusú előjeles és előjel nélküli adatokon.

5.7.1 Bitenkénti logikai műveletek

A műveletek első csoportja, a bitenkénti logikai műveletek, lehetővé teszik, hogy biteket teszteljünk, töröljünk vagy beállítsunk:

<i>Operátor</i>		<i>Művelet</i>
~	!	1-es komplementum
&		bitenkénti <i>ÉS</i>
		bitenkénti <i>VAGY</i>
^		bitenkénti kizáró <i>VAGY</i>

A bitenkénti logikai műveletek működésének leírását az alábbi táblázat tartalmazza, ahol a 0 és az 1 számjegyek a törölt, illetve a beállított bitállapotot jelölik.

<i>a</i>	<i>b</i>	<i>a & b</i>	<i>a b</i>	<i>a ^ b</i>	<i>~a</i>
1	1	1	1	0	0
1	0	0	1	1	0
0	1	0	1	1	1
0	0	0	0	0	1

A bitenkénti logikai műveletek a C++ nyelv szintjén biztosítják a számítógép hardver-eleminek programozását. A perifériák többségének alacsony szintű vezérlése bizonyos bitek beállítását, illetve törlését jelenti. Ezeket a műveleteket összefoglaló néven „maszkolásnak” nevezzük.

Minden egyes művelethez megfelelő bitmaszkot kell készítenünk, amellyel aztán logikai kapcsolatba hozva a megváltoztatni kívánt értéket, végbemegy a kívánt bitművelet. Mielőtt sorra vennénk a szokásos bitműveleteket, meg kell ismerkednünk az 1-, 2- és 4-bájtos adatelemek bitjeinek sorszámozásával. A bitek sorszámozása mindig a legkisebb helyértékű bittől indulva 0-val kezdődik, és balra haladva növekszik. Mivel a példáinkban **short int** típusú adatelemeket használunk, nézzük meg ezen adatok felépítését:

15.	14.	13.	12.	11.	10.	9.	8.	7.	6.	5.	4.	3.	2.	1.	0.
0	0	0	0	0	1	1	1	1	1	0	1	0	0	1	1

A fenti szám értéke hexadecimális szárendszerben 0x07D3 illetve decimális alakban 2003.

Az alábbi példákban a `short int` típusú 2525 szám 4. és 13. bitjeit kezeljük:

`short int a = 2525; // 0x09da`

Művelet	Maszk	C++ utasítás	Eredmény
Bit 1-be állítása	0010 0000 0001 0000	<code>a = a 0x2010;</code>	0x29da
Bitok törlése	1101 1111 1110 1111	<code>a = a & 0xdfe;</code>	0x09cd
Bitok negálása	0010 0000 0001 0000	<code>a = a ^ 0x2010;</code>	0x29cd (10701)
Az összes bit negálása		<code>a = ~a;</code>	0xf622 (2525)

Felhívjuk a figyelmet a kizáró vagy operátorra (^). Ha ugyanazzal a maszkkal kétszer végezzük el a műveletet, visszakapjuk az eredeti számot, esetünkben a 2525-öt.

5.7.2 Biteltoló műveletek

A bitműveletek másik csoportjába, a biteltoló (*shift*) operátorok tartoznak. Az eltolás balra (<<) és jobbra (>>) egyaránt elvégezhető. Az eltolás során a bal oldali operandus bitjei annyiszor lépnek balra (jobbra), amennyi a jobb oldali operandus értéke. A fel-

`short int a;`

Értékkadás	Bináris érték	Művelet	Eredmény decimális (hexa) bináris
<code>a=2525;</code>	0000 1001 1101 1101	<code>a=a<<2;</code>	10100 (0x2774)
<code>a=2525;</code>	0000 1001 1101 1101	<code>a=a>>3;</code>	315 (0x013b)
<code>a=-2525;</code>	1111 0110 0010 0011	<code>a=a>>3;</code>	-316 (0xfec4)
<code>a=-2525;</code>	1111 0110 0010 0011		1111 1110 1100 0100

Az eredményeket megvizsgálva láthatjuk, hogy az 1 bittel való balra eltolás során az *a* változó értéke kétszeresére (2^1) nőtt, míg két lépéssel jobbra eltolva, *a* értéke negyed (2^2) részére csökkent. Általánosan is megfogalmazható, hogy valamely egész szám biteinek *n* lépéssel történő balra tolása a szám (2^n) értékkel való megszorzását ered-
ményezi. Az *m* bittel való jobbra eltolás pedig (2^m) értékkel elvégzett egész osztásnak felel meg.

Írjunk egy olyan programot, amely a beolvasott 16-bites egész számot két bájtira bontja! A feladat aritmetikai és bitenkénti műveletek felhasználásával egyaránt megoldható. Nézzünk egy lehetséges megoldást, amelyben bitműveleteket használunk!

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

```
void main()
```

```
{
```

```
    short int num;
```

```
    unsigned short lo, hi;
```

```
    // A szám beolvasása
```

```
    cout<<"\nKérek egy egész számot [-32768,32767] : ";
```

```
    cin>>num;
```

```
    cout<<"A szám = "<<num<<endl;
```

```
    // Az alsó bájt meghatározása maszkolással
```

```
    lo=num & 0x00FF;
```

```
    cout<<"A szám alsó bájtja: "<<lo<<endl;
```

```
    // Az felső byte meghatározása biteltolással
```

```
    hi=num >> 8;
```

```
    cout<<"A szám felső bájtja: "<<hi<<endl;
```

```
    cin.get();
```

```
}
```

5.8 Értékkadó operátorok

Már említettük, hogy C++ nyelvben az értékkadás egy olyan kifejezés, amely a bal oldali operandus által kijelölt tárolónak adja a jobb oldalon megadott kifejezés értékét, másrészt pedig ez az érték egyben az értékkadó kifejezés értéke is. Ebből következik, hogy értékkadás tetszőleges kifejezésben szerepelhet.

Ha az a és b int típusú változók, akkor az értékkadás hagyományos formái

```
a = 23;
b = (a+12)*4-9;
```

során az a változó értéke 23, míg a b változóé 131 lesz. Felírható azonban olyan, más nyelvektől idegen kifejezés is,

```
b=2*(a=4)-5;
```

ahol az a (4) és b (3) változók egyaránt értéket kapnak.

Az értékadó operátorok kiértékelése jobbról-balra haladva történik. Emiatt C++ nyelv-
ven használható a többszörös értékadás, melynek során több változó veszi fel ugyanazt
az értéket:

$$a = b = 23;$$

Az értékadások gyakran használt formája, amikor egy változó értékét valamilyen mű-
velettel módosítjuk, és a keletkező új értéket tároljuk a változóban:

$$a = a + 2;$$

Az ilyen alakú kifejezések tömörebb formában is felírhatók:

$$a += 2;$$

Altalában elmondható, hogy a

$$k1F_1 = k1F_1 \text{ op } k1F_2$$

alakú kifejezések felírására az ún. összetett értékadás műveletét is használhatjuk:

$$k1F_1 \text{ op } = k1F_2$$

A két felírás egyenértékű, attól a különbségtől eltekintve, hogy a második esetben a
bal oldali kifejezés kiértékelése csak egyszer történik meg. Operátorként (*op*) az eddig
megismert kétoperandusú műveleteket használhatjuk:

<i>Hagyományos forma</i>	<i>Tömör forma</i>
$a = a + b$	$a += b$
$a = a - b$	$a -= b$
$a = a * b$	$a *= b$
$a = a / b$	$a /= b$
$a = a \% b$	$a \% = b$
$a = a << b$	$a << = b$
$a = a >> b$	$a >> = b$
$a = a \& b$	$a \& = b$
$a = a b$	$a = b$
$a = a \wedge b$	$a \wedge = b$

Meg kell jegyeznünk, hogy az

$$a *= b + 1;$$

kifejezésnek megfelelő hagyományos értékadás az

$$a = a * (b + 1);$$

nem pedig az

```
a = a * b + 1;
```

Az összetett értékadás használatát általában gyorsabb kódot eredményez, és könnyebben értelmezhetővé teszi a forrásprogramot.

5.9 Pointerműveletek

A C++ nyelvben található két olyan speciális egyoperandusú művelet, amelyeket műveletökkel kapcsolatban alkalmazunk. A „cím” (&) művelet eredménye az operandusként megadott tároló címe:

```
int a = 23, *ptr;  
ptr = &a;
```

A „cím” operátort arra használjuk, hogy mutatóinkat már meglévő változókra irányítsuk. A másik mutatóoperátor (*) az indirekt hivatkozás elvégzéséhez szükséges:

```
*ptr = *ptr + 5;
```

A *ptr kifejezés a ptr pointer által mutatott tárolót jelöli.

A mutatókkal a fentiekben túlmenően további (aritmetikai) műveleteket is végezhetünk. A pointer léptetése a szomszédos elemre többféle módon is elvégezhető:

```
int *p, *q;  
...  
p = p + 1;  
p += 1;  
p++;  
++p;
```

Az előző elemre való visszalépésre szintén több lehetőség közül választhatunk:

```
p = p - 1;  
p -= 1;  
p--;  
--p;
```

A két mutató különbsége, vagyis a két mutató között elhelyezkedő elemek száma szintén meghatározható:

```
int h = p - q;
```

A léptető és az indirekt operátorok együttes használatához kellő óvatosság szükséges. (Az alábbi példában az sp mutatóval egy konstans karaktersorozatra mutatunk.)

```
char *sp = "Lafentita";
```

A definíció után az *sp* mutató a sztring első karakterére, az *'L'* betűre mutat. Mi lesz a helyzet a

```
cout<<"+*sp<<endl;
cout<<"sp<<endl;
```

utasítások végrehajtása után? A precedencia-táblázatban azt találjuk, hogy az indirekt-ség és az előrevetett lépető operátorok azonos precedenciával rendelkeznek. A kérdés megválaszolásánál az asszociativitási szabályt, a kiértékelés jobbról-balra történő menetét kell figyelembe vennünk. Ennek alapján először az indirekt-ség operátort értelmelmezi a rendszer. Ezt követi a léptes operátorának feldolgozása, melynek hatására az első kiírás során az *'M'* betű jelenik meg, míg a második utasítással a *„Mafentia”* karakterlánc lesz látható.

A *** és a *++* operátorok lehetséges elrendezéseit, amelyek a fenthez hasonló gondolatmenettel értelmezhetők, táblázatba rendeztük. Az első oszlop a kifejezéseket tartalmazza, míg a második és a harmadik oszlopban a két kiírás eredménye látható.

Kifejezés	Karakter	Szöveg
*++sp	a	a
*sp++	L	a
++*sp	M	a
(*sp)++	L	a
++*sp++	M	a
++*sp++	L	a
++*sp++	M	a

5.10 A sizeof operátor

A C++ nyelv tartalmaz egy olyan fordítás idején kiértékelésre kerülő egyoperandusú operátort, amely tetszőleges változó, típus, kifejezés méretét megadja. A

sizeof változó

sizeof (típusnév)

alakú kifejezések értéke egy olyan egész szám, amely megegyezik a megadott változó, illetve típus bajtban kifejezett méretével.

A változó tetszőleges egyszerű változó, tömb vagy objektum egyaránt lehet:

```
int a;
double d;
```

```
a = sizeof d;
```

```
// A d változó mérete
```

A típusnev az alaptípusokon túlmenően tetszőleges származtatott típust is jelölhet:

```
a = sizeof (double); // A double típus mérete
a = sizeof (char *); // A mutató típus mérete
```

5.11 A vessző operátor

Egyetlen kifejezésben több, akár egymástól független kifejezés is elhelyezhető, a vessző operátor felhasználásával. A vessző operátor tartalmazó kifejezés balról-jobbra haladva értékelődik ki, a kifejezés értéke és típusa megegyezik a jobb oldali operandus értékével, illetve típusával.

Példaként tekintsük az

```
x = ( y = 2 , y + 3 );
```

kifejezést! A kiértékelés a zárójelbe helyezett vessző operátorral kezdődik, melynek során először az *y* változó kap értéket (2), majd pedig a zárójelzett kifejezés (2+3=5). Végezetül az *x* változó értékadással megkapja az 5 értéket.

A vessző operátor gyakran használjuk különböző változók kezdőértékének egyetlen utasításban (kifejezésben) történő beállítására:

```
int x, y;
double z;
```

```
x = 12, y = 23, z = 1.2345 ;
```

Ugyancsak a vessző operátor kell alkalmaznunk, ha két változó értékét egyetlen utasításban kívánjuk felcserélni (harmadik változó felhasználásával):

```
int a=12, b=23, c;
```

```
c = a, a = b, b = c;
```

Felhívjuk a figyelmet arra, hogy azok a vesszők, amelyeket a deklarációkban a változónevek, illetve a függvényhívások az argumentumok elkülönítésére használunk *nem* vessző operátorok. Ezért ezekben az esetekben nem garantált a balról-jobbra haladó kiértékelési sorrend.

5.12 A feltételes operátor

A feltételes operátor (?) három operandussal rendelkezik:

```
kif1 ? kif2 : kif3
```

A feltételes kifejezésben először a kif_1 kifejezés kerül kiértékelésre. Amennyiben ennek értéke nem nulla (igaz), akkor a kif_2 értéke adja a feltételes kifejezés értékét. Ellenkező esetben a kettőspont után álló kif_3 értéke lesz a feltételes kifejezés értéke. Így módon a kettőspont két oldalon álló kifejezések közül mindig csak az egyik értékelődik ki. A feltételes kifejezés típusa a nagyobb pontoságú részkifejezés típusával egyezik meg. Az

```
(n > 0) ? 3.141534 : 54321L;
```

kifejezés típusa, függetlenül az n értéktől mindig **double** lesz. A feltételes operátort a legkülönbözőbb célokra használhatjuk. Az esetek többségében a feltételes utasítást (**if**) helyettesítjük vele. A következő két példában az a és b értékek közül kiválasztjuk a nagyobbbat.

Megoldás az **if** utasítás felhasználásával:

```
if (a > b)
    z = a;
else
    z = b;
```

Feltételes kifejezéssel sokkal tömörebben oldható meg a feladat:

```
z = a > b ? a : b;
```

Nézzünk két jellegzetes példát a feltételes operátor alkalmazásáról! Az első esetben a ch karakter kitérítendő formátumot feltevelés kifejezéssel adjuk meg. Ha a ch karakter vezérlő karakter (kódja < 32), akkor a hexadecimalis kódját, ellenkező esetben pedig magát a karaktert írjuk ki:

```
printf("ch < 32 ? \"%02X\\n\" : \"%2c\\n\", ch);
```

Az alábbi kifejezés segítségével a 0 és 15 közötti értékeket hexadecimalis számjeggyé alakíthatjuk:

```
ch = n >= 0 && n <= 9 ? '0' + n : 'A' + n - 10;
```

5.13 Az érvényességi kör (hatókör) operátor

Az érvényességi kör operátor kettős szerepet tölt be a C++ nyelvben. A :: operátor segítségével a program tetszőleges blokkjából hivatkozhatunk a globális (fájliszintű) hatókörrel rendelkező nevekre.


```

int i = 1223;
void main() {
    double i = 3.14159265;
}
long i = 2, a;
a=i*::i; // az a változó értéke 2*1223=2446
::i=94; // az ::i változó értéke 94
}

```

Az erőnyességi kör operátor használjuk akkor is, amikor valamely osztály adattagjaira, illetve tagfüggvényeire hivatkozunk. Ugyancsak ezt az operátort alkalmazzuk, ha valamely névtérületen definiált neveket a **using namespace** deklaráció nélkül kívánjuk elérni:

```

#include <iostream>

void main()
{
    std::cout<<"C++ nyelv"<<std::endl;
    std::cin.get();
}

```

5.14 A new és a delete operátorok használata

A szabad memória dinamikus használata alapvető részét képezi minden C++ nyelvén megírt programnak. A C programokban könyvtári függvényekkel végezhetjük el a szükséges memóriafoglalást (*malloc()*,...) illetve felszabadítást (*free()*) műveleteket. A C++ nyelvben a **new** és **delete** operátorok nyelvddefiníció szintjén helyettesítik a fenti könyvtári függvényeket.

A **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal a szabad memóriában, és a területre mutató pontint ad eredményül:

```

int * ip;
ip = new int;

```

A **delete** operátor a **new** operátor által lefoglalt területet felszabadítja:

```
delete ip;
```

A **new** segítségével nem csak egyetlen elemnek, hanem több egymás után elhelyezkedő elemnek is helyet foglalhatunk a memóriában. Ez így létrejövő adatstruktúrát dinamikus tömbnek nevezzük. (A hagyományos, statikus tömböket könyvünk 7. fejezetében tárgyaljuk.)

Nézzünk néhány példát a **new** művelet használatára! Először foglaljunk tárterületet egy 10-elemű egész tömb számára!

```
int *ap;
ap = new int [10];
```

A `lefoglalt` terület felszabadítására a `delete[]` operátort kell használnunk:

```
delete[] ap;
```

Nézzük meg, mi változik, ha 10-elemű egészsre mutató pointertömböt kell dinamikusan létrehoznunk!

```
int **pa;
pa = new int * [10];
...
delete[] pa;
```

A memóriafoglalásnál, főleg amikor nagy méretű dinamikus tömb számára kívánunk tárt területet foglalni, előfordulhat, hogy nem áll rendelkezésünkre elegendő memória. A C++ futtatórendszer ezt a helyzetet a ***bad_alloc*** kivétellel (*exception*) létrehozásával jelzi. Így a kivételkezelés eszköztárát használva tehetjük programunkat biztonságossá.

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

void main() {
    long *data, size;
    cout << "\nKérem a tömb méretét: ";
    cin >> size;
    cin.get();
    // Memóriafoglalás
    try {
        data = new long [size];
    }
    catch (bad_alloc) {
        // Sikertelen foglalás
        cerr << "\nNincs elég memória!\n" << endl;
        cin.get();
        exit(-1); // Kilépünk a programból
    }
    // ...
    delete[] data;
    cin.get();
}
```

Amennyiben nem kívánunk elni a kivételkezelés adta lehetőségeikkel, a `new` operátor után meg kell adnunk a ***nothrow*** memóriafoglalót, melynek hatására az operátor a kivétel létrehozása helyett 0 értékkel tér vissza.

```

#include <iostream>
using namespace std;

void main() {
    long *data, size;
    cout << "\nKérem a tömb méretét: ";
    cin >> size;
    cin.get();

    // Memóriafoglalás
    data = new (nothrow) long [size];
    if (!data) {
        cerr << "\nNincs elég memória!\n" << endl;
        cin.get();
        exit(-1); // Kilépünk a programból
    }

    // ...

    // A lefoglalt memória felszabadítása
    delete[] data;
    cin.get();
}

```

A *malloc()* és a *free()* függvények kompatibilitási megfontolások miatt továbbra is elérhetők a C++ nyelvből, de helyettük ajánlott a **new** és a **delete** operátorokat alkalmazni. Ha a **new** és a **delete** operátorokkal foglalnunk helyet valamely osztályobjektum számára, akkor a rendszer automatikusan meghívja az osztály konstruktorát a **new**, illetve a destruktort a **delete** művelet végrehajtásakor.

További kellemes lehetősége a C++ nyelvnek, hogy sikertelen memóriafoglalás esetén egyetlen kezelőfüggvény bevezetésével elvégezhető a szükséges lépések (például üzenet kirírása). Ehhez a *set_new_handler()* függvény segítségével kell az új kezelő-

```

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

// A new operátor új hibakezelője:
void mem_kezelo() {
    cerr << "\nNincs elég memória!";
    cin.get();
    exit(1); // kilépés a programból
}

```

```
void main(void)
{
    set_new_handler(mem_kezelő); // Az új hibakezelő beállítása
    char* ptr = new char[1000000];
    cout << "Nélsz memóriafoglalás: ptr = "<<hex<<long(ptr);
    ptr = new char[1900000000];
    cout << "Nélsz memóriafoglalás: ptr = "<<hex<<long(ptr);
    // ...
    cin.get();
    set_new_handler(0); // Az eredeti hibakezelés visszaállítása
}
```

Tudnunk kell azonban, hogy a *mem_kezelő()* függvényben elvégzett tevékenység után (például memória-felszabadítás) újra végrehajtodik a memóriafoglalás. Ennek elkerülése érdekében az *exit()* függvény hívásával kilépünk a programból.

A kifejezet végén a *new* operátor további lehetőségeire kívánjuk felhívni a figyelmet. A *new* után közvetlenül egy zárójelben megadott mutató is állhat, melynek hatására az operátor a mutató címével tér vissza (vagyis nem foglal memóriát):

```
int* p=new int(12);
int* q=new(p) int(23);

char* s1="Nata";
char* s2=new(s1) char;
```

A fenti példákban a *q* a *p* által mutatót területre, míg az *s2* az *s1* karaktertetsorozatra hivatkozik. A mutatók eltérő típusúak is lehetnek:

```
long a=0x19792003;
char* p=new(&a) char;
```

5.15 Futásidejű típusazonosítás

C++-ban a *typeid* operátor egy *type_info* típusú objektum (*typeinfo*) referenciáját adja vissza. A *type_info* objektum az operandus típusáról tartalmaz információkat (*Runtime Type Information*).

```
typeid (kifejezés)
typeid (típusazonosító)
```

A *typeid* operátor segítségével futásidejű típusazonosítást végezhetünk. (Hibás esetben *bad_typeid* kivételt keletkeztet.)

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

...
if (typeid(A) == typeid(B))
    cout << "A és B típusa: " << typeid(A).name();
```

5.16 Típuskonverziók

A kifejezések kiértékelése során előfordulhat, hogy valamely kétoperandusú operátor különböző típusú operandusokkal rendelkezik. Ahhoz azonban, hogy a művelet elvégezhető legyen, a fordítónak azonos típusúra kell átalakítania a két operandust, vagyis típuskonverziót kell végrehajtania.

A típuskonverziók egy része automatikusan, a programozó beavatkozása nélkül megy végbe, a C++ nyelv definíciójában rögzített szabályok alapján. Ezeket a konverziókat *implicit* vagy *automatikus* konverzióknak nevezzük.

Explicit típuskonverziót azonban a programozó is előírhat a C++ programban, a típuskonverziós operátorok (*cast*) felhasználásával.

5.16.1 Implicit típuskonverziók

Általánosságban elmondható, hogy az automatikus konverziók során a „szűkebb” operandus információvesztés nélkül konvertálódik a „szélesebb” operandus típusára. Az alábbi kifejezés kiértékelése során az `int` típusú *i* operandus `float` típusú lesz, ami egyben a kifejezés típusát is jelenti:

```
int i=20, j;
float f=3.45;
i + f;
```

Az implicit konverziók azonban nem minden esetben mennek végbe információ-vesztés nélkül. Az értéktadás és a függvényhívás paraméterezése során tetszőleges típusok közötti konverzió is előfordulhat. Ha a fenti példában az összeget a *j* változónak feleltjük meg, vagy argumentumként adjuk az *abs()* függvénynek

```
j = i + f;
```

akkor bizony adatavesztés lép fel, hiszen az összeg törtrésze elvesz, és *23* lesz a *j* változó, illetve a függvényparaméter értéke.

A következőkben röviden összefoglaljuk az *x op y* alakú kifejezések kiértékelése során automatikusan végrehajtott konverziókat.

1. A `char`, a `wchar_t`, a `short`, a `bool` és az `enum` (felsorolt) típusú adatok automatikusan `int` típusra konvertálódnak. Ha az `int` típus nem alkalmas az értékek tárolására, akkor **unsigned int** lesz a konverzió cél típusa. Ez a konverziós szabály az „egész konverzió” (*integral promotion*) nevet viseli. Mivel a fenti konverziók érték- és előjelhelyes eredményt adnak, értékmegőrző konverzióknak is szokás nevezni azokat.

2. Ha az első lépés után a kifejezésben különböző típusok szerepelnek, életbe lép a típusok hierarchiája szerinti konverzió:

```
int < unsigned < long < unsigned long < long long < unsigned long long <
float < double < long double
```

A típusátalakítás során a „kisebb” típusú operandus a „nagyobb” típusúvá konvertálódik. Az átalakítás során felhasznált szabályok a „szokásos aritmetikai konverziók” nevet viselik.

A C++ implicit mutatókonverzióval is rendelkezik. Tetszőleges típusú mutató átalakítható általános (**void***) mutatótípussá. Ellenkező irányú konverzióhoz explicit típusátalakítást kell használnunk. Ugyancsak érdemes megjegyezni, hogy a nulla (0) számértékekkel tetszőleges típusú mutató inicializálható.

```
int x=5;
void * p=&x;
int *q=0;
q=static_cast<int*>(p);
cout<<<<endl;
```

5.16.2 Explicit típusátalakítások

A fordítás közben végrehajtott (statikus), gyakran használt átalakítások kijelölésére több lehetőség közül is választhatunk:

```
típusátalakítás (típusnév) kifejezés (char *) p
érték leírhezoása típusnév (kifejezés) int (a)
```

```
ellenőrzött típusátalakítás static_cast<típusnév>(kifejezés) static_cast<char *>(p)
```

Vegyük sorra a szabványos C++ nyelv típusátalakítási lehetőségeit, amelyek finomítják a korábbi C++ változatok típusmódosító megoldásait!

Konstans típusátalakítás

```
const_cast < típus > (arg)
```

A **const_cast** operátort akkor használjuk, ha fel kívánjuk oldani a **const** és/vagy **volatile** típusmódosítók hatását egy adott változó esetén. Ez a típusmódosítás fordítási időben történik. A **const_cast** használatakor a *típus* és az *arg* azonos típusúak kell legyenek. Az alábbi példában egy nem konstans **int** argumentumot váró függvényt hívunk **const** argumentummal:

```
const int cvalo=10;
fv(const_cast<int*>(cvalo));
```

Dinamikus típusátalakítás

```
dynamic_cast < típus > (ptr)
```

A dinamikus típusátalakítással futásidejű konverziókat végezhetünk. Ha a típusátalakítás nem hajtható végre, akkor a kifejezés 0 értékű lesz, és *bad_cast* kivételt jön létre. A *típus* osztályra mutató pointer, referencia vagy **void*** típusú kifejezés, míg a *ptr* operandus pointer vagy referencia típusú kifejezés lehet.

Fontos megjegyeznünk, hogy a **dynamic_cast** végrehajtásához ún. futásidejű típusazonosításra (*run time type identification* - RTTI) van szükség.

„Veszélyes” típusátalakítások

```
reinterpret_cast < típus > (arg)
```

A kifejezésben a *típus* mutató, referencia, numerikus típus, függvénymutató vagy oszthatósági mutató, illetve egész típusú kifejezés mutatóvá alakítható. Két inkompatibilis mutatótípus közötti konverzió elvégzésére is használható a **reinterpret_cast**.

```
int i = reinterpret_cast<int> (&x);
```

Statikus típusátalakítások

```
static_cast < típus > (arg)
```

A **static_cast** segítségével jól definiált, hordozható és visszafordítható típuskonverziókat hajthatunk végre. Mind a *típus*, mint pedig az *arg* típusának fordítási időben ismertnek kell lennie.

```
char ch = static_cast<char> ('A'+1.0);
```

Mivel a fenti esetekben a típusnév megjelenik a konverziós előírásban, *explicit* típuskonverzióról beszélünk.

5.17 Bővebben a konstansokról

A C nyelvben létezik ugyan a **const** definíció, de valójában ez is egy változó, amelyhez nem enged közvetlen hozzáférést a fordító (közvetett módon igen).

```
const int ci = 1979;
int * pi;

ci = 2005; // Hiba C-ben és C++-ban
pi = &ci; // csak C++-ban hiba
*pi = 2005;
```

A C++ fordító sokkal szigorúbban ellenőrzi a **const** típusú konstansok felhasználását:

```
const double pi = 3.14159265; // szabályos konstansdefiniáció
```

Konstansra csak megfelelő mutatóval hivatkozhatunk:

```
const double ac = 12.23;
double * pd = &ac; // Hiba!

const double *pdc;
const double dc = 12.23;
// double konstansra mutató pointer

pdc = &dc;
// a pdc pointer a dc-re mutat
pdc = &d;
// A pointer segítségével a d változó sem változtatható meg.
pdc = 9.4; // Hiba!
```

Konstans értékű pointer:

```
int ev;
int * const aktev = &ev;
// Az aktev pointer értéke nem változtatható meg,
// de a *aktev módosítható!
aktev = 2005;
aktev = &ev; // Hiba!
```

Konstansra mutató konstans értékű pointer:

```
const int ev = 1979;
const int ae = 2005;
// int típusú konstansra mutató konstans pointer
const int * const aktev = &ev;
aktev = &ae; // Hiba!
aktev = 2525; // Hiba!
```

Ugyancsak a **const** típusú konstansok használata mellett szól az a lehetőség, hogy statikus tömbök definíciójában is felhasználhatók:

```
const maximum = 100;
int num[maximum];
```


6. A C++ nyelv utasításai

A C++ nyelven megírt program végrehajtható része elvégzendő tevékenységekből (utasításokból) épül fel. Az utasítások a strukturált programozás alapelveinek megfelelően ciklusok, programelágazások és vezérlésátadások szervezését teszik lehetővé. A C++ nyelv más nyelvekhez hasonlóan rendelkezik a vezérlésátadás **goto** utasításával, melynek használata nehezen követhetővé teszi a program szövegét. A **goto** utasítás alkalmazása azonban az esetek többségében elkerülhető a **break** és a **continue** utasítások bevezetésével. A C++ nyelv utasításait hét csoportba sorolhatjuk:

Kifejezés utasítás	Ures utasítás:
Osszetett utasítás:	{ }, try { }, catch { }
Szelekciós utasítások:	if, else, switch
Címkezett utasítások:	case, default, <i>ugrdási címke</i>
Vezérlésátadó utasítások:	break, continue, goto, return,
Iterációs (ciklus) utasítások:	do, for, while

Az utasítások részletes tárgyalásánál nem a fenti csoportosítást követjük, hiszen az egyes utasítások használatakor különböző csoportokba elhelyezkedő utasításokat kell alkalmaznunk.

6.1 Utasítások és blokkok

Tetszőleges kifejezés utasítás lesz, ha pontosvesszőt (;) helyezünk mögé:

kifejezés;

A *kifejezés utasítás* végrehajtása a kifejezésnek az 5. fejezetben ismertetett szabályok szerint történő kiértékelését jelenti. Mielőtt a következő utasításra kerülne a vezérlés, a teljes kiértékelés (a mellékhatásokkal együtt) végbe megy. Nézzünk néhány kifejezés utasítást:

```
x = y + 3; // értékadás
x++; // az x növelése 1-gyel
x = y = 0; // többszörös értékadás
fv(arg1, arg2); // void típusú függvény hívása
y = z = f(x) + 3; // függvényt hívó kifejezés
```

Az *ures utasítás* egyetlen pontosvesszőből (;) áll. Használataira akkor van szükség, amikor logikailag nem kívánunk semmilyen tevékenységet végrehajtani, azonban a szintaktikai szabályok szerint a program adott pontján utasításnak kell szerepelnie. Az

üres utasítást, melynek végrehajtásakor semmi sem történik, gyakran használjuk a **do**, **for**, **while** és **if** szerkezetekben.

A kapcsos zárójelek ({ }) segítségével a logikailag összefüggő deklarációkat és utasításokat egyetlen *utasításba* vagy *blokkba* csoportosíthatjuk. Az összetett utasítás mindenütt felhasználható, ahol egyetlen utasítás megadását engedélyezi a C++ nyelv leírása. Összetett utasítást, melynek általános formája:

```
{
    lokális definíciók, deklarációk és
    utasítások
}
```

a következő három esetben használunk:

1. Amikor több logikailag összefüggő utasítást egyetlen utasításként kell kezelni (ilyenkor általában csak utasításokat tartalmaz a blokk),
2. Függvények törzseként,
3. Definíciók és deklarációk érvényességének lokalizálására.

Az utasításblokkon belül az utasításokat és a definíciókat/deklarációkat tetszőleges sorrendben megadhatjuk. Felhívjuk a figyelmet arra, hogy blokkot nem kell pontosvesszővel lezárni.

A következő példa összetett utasításában felcseréljük az *a* és *b* változók értékét, amennyiben *a* nagyobb, mint *b*:

```
int a, b;
if ( a > b )
{
    int c = a;
    a = b;
    b = c;
}
```

6.2 Az if utasítás

A C++ nyelv két lehetőséget biztosít a program kódjának feltételhez kötött végrehajtására – az **if** és a **switch** utasításokat. Az **if** utasítás segítségével valamely tevékenység (*utasítás*) végrehajtását egy *kifejezés* (feltétel) értékétől tehetjük függővé. Az **if** alábbi formájában az *utasítás* csak akkor hajtodik végre, ha a *kifejezés* értéke nem nulla (igaz, **true**):

```
if ( kifejezés )
    utasítás
```

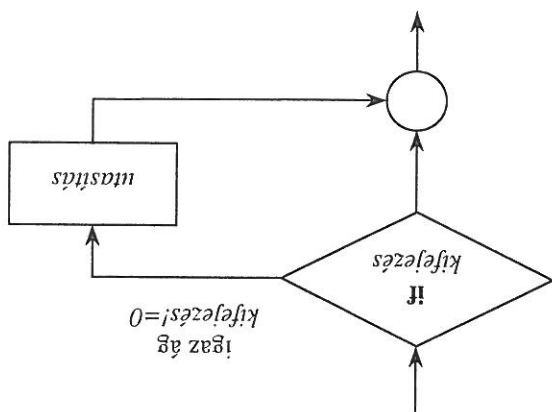
A következő példaprogram egyetlen karaktert olvas a billentyűzetről az `<Enter>` gombbal lezárva. Ha a karakter az `escape (<Esc>)`, akkor kilépés előtt hangjelzést ad, ellenkező esetben a program egyszerűen befejezi a futását.

```
#include <iostream>
using namespace std;
const int ESC = 27;

void main(void)
{
    char ch;
    cout<<"Kerek egy karaktert: ";
    ch=cin.get();
    if (ch == ESC)
        cout<<"\aEsc"<<endl;
}
```

A különböző vezérlési szerkezetek működésének grafikus szemléltetésére blokk-diagramot szokás használni. Az egyszerű `if` utasítás feldolgozását az alábbi ábrán kö-

vethejük nyomon:



Mivel az `if` utasítás feltétele egy numerikus kifejezés nem nulla voltának tesztelése, a kód kézenfekvő módon egyszerűsíthető, ha az

```
if (kifejezés != 0)
```

helyett az

```
if (kifejezés)
```

formát használjuk. Ez a megoldás általában világos, de bizonyos esetekben rejtelyesnek tűnhet. Külön felhívjuk a figyelmet arra, hogy a feltétel-kifejezést körülvevő záró-

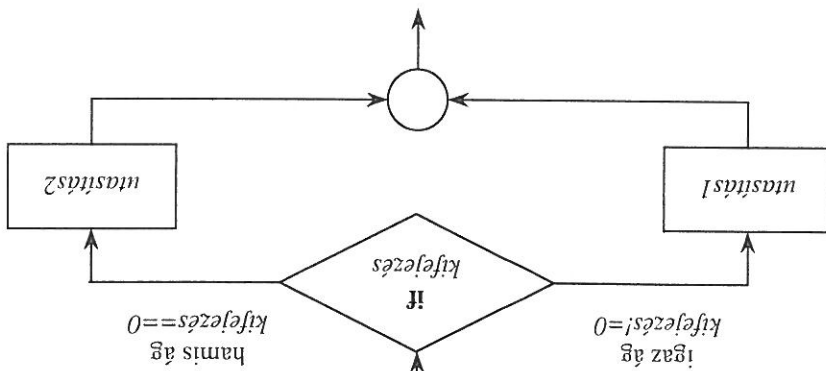
jelet mindig ki kell tenni.

6.2.1 Az if-else szerkezet

Az if utasítás teljes formájában, amely tartalmazza az else-ágot, arra az esetre is megadhatunk egy tevékenységet (*utasítás2*), amikor a *kifejezés* (feltétel) értéke zérus (hamis, **false**). Ha az *utasítás1* és az *utasítás2* nem összetett utasítások, akkor pontos vesszővel kell őket lezárni.

```
if (kifejezés)
    utasítás1
else
    utasítás2
```

Az if-else konstrukció logikai vázlata a következő ábrán látható.



Az alábbi példában a beolvasott egész számról if utasítás segítségével döntjük el, hogy páros vagy páratlan:

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

void main()
{
    cout<<"Kérek egy egész számot: ";
    int n;
    cin>n;
    cin.get();
    if (n % 2 == 0)
        cin.get();
    else
        cout<<"A szám páros!"<<endl;
    cout<<"A szám páratlan!"<<endl;
    cin.get();
}
```

Az if utasítások egymásba is ágyazhatók. Ilyenkor azonban óvatosan kell eljárniuk, hisz a fordító nem mindig úgy értelmezi az utasítást, ahogy mi gondoljuk. Az alábbi!

példában azt várjuk az **if**-es szerkezettől, hogy a megadott egész számról megmondja, hogy az negatív páros szám-e, vagy nem negatív szám. A megoldás

```
if (n < 0)
    if (n % 2 == 0)
        cout<<"Negatív páros szám."<<endl;
    else
        cout<<"Nem negatív szám."<<endl;
```

azonban nem működik helyesen, hiszen a negatív páratlan számokat is nem negatív-nak mondja. Hol a hiba? A példában az **else**-ágot a külső **if**-hez kívántuk kapcsolni, azonban a fordító minden **if** utasításhoz a hozzá legközelebb eső **else** utasítást rendeli. A helyes működéshez kétféleképpen is eljuthatunk. Az egyik lehetőség, ha a belső **if** utasításhoz egy üres utasítást (;) tartalmazó **else**-ágot kapcsolunk:

```
if (n < 0)
    if (n % 2 == 0)
        cout<<"Negatív páros szám."<<endl;
    else ;
    else
        cout<<"Nem negatív szám."<<endl;
```

A másik járható út, ha a belső **if** utasítást kapcsolos zárójelek közé, azaz utasításblokkba helyezzük:

```
if (n < 0) {
    if (n % 2 == 0)
        cout<<"Negatív páros szám."<<endl;
    else
        cout<<"Nem negatív szám."<<endl;
```

6.2.2 Az else-if szerkezet

Az egymásba ágyazott **if** utasítások gyakran használt formája, amikor az **else**-ágakban szerepel az újabb **if** utasítás:

```
if (Kifejezés)
    utasítás
else if (Kifejezés)
    utasítás
else if (Kifejezés)
    utasítás
else
    utasítás
```

Ezzel a szerkezettel a program többirányú elágaztatását valósíthatjuk meg. Ha bármelyik *kifejezés* igaz, akkor a hozzákapcsolódó *utasítás* hajtódik végre

Amennyiben egyik *feltétel* sem teljesült, a program végrehajtása az utolsó **else** utasítással folytatódik. Bizonyos esetekben nincs szükség alapértelmezés szerinti tevékenység végrehajtására – ekkor az

```
else
    utasítás
```

elhagyható. Külön kiemelés érdemel az **else-if** programtervezlet olvashatósága. Az alábbi példában az n számról eldöntjük, hogy az negatív, nulla, vagy pozitív:

```
if (n > 0)
    cout<<"Pozitív szám"<<endl;
else if (n==0)
    cout<<"Nulla"<<endl;
else
    cout<<"Negatív szám"<<endl;
```

Az **else-if** szerkezet speciális esete, amikor a felhasználó kifejezések egyenlőség-vizsgálátokat ($==$) tartalmaznak. Az alábbi példában egyszerű kalkulátort valósítottunk meg, amely a négy alapművelet elvégzésére képes. A program indítása után a kívánt műveletet például

12.23+34.45

alakban kell megadni. A programban nem vizsgáljuk nullával való osztást, ennek kezeltését a futtató rendszerre bíztuk.

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

const char PROMPT = " : ";

void main()
{
    double a,b,e;
    char op;
    cout.put(PROMPT);
    cin.get();
    if (op == '+')
        e = a + b;
    else if (op == '-')
        e = a - b;
    else if (op == '*')
        e = a * b;
    else if (op == '/')
        e = a / b;
    // a készenléti jel
    // beolvasás
    // összeadás?
    // kivonás?
    // szorzás?
    // osztás?
```

```

else {
    cerr<<"Hiba: művelet!"<<endl;
    cin.get();
    return;
}
cout<<"a<op>b<<"<<endl;
cin.get();
}

```

6.3 A switch utasítás

A **switch** utasítás többirányú programelágaztatást tesz lehetővé olyan esetekben, amikor egy egész kifejezés értékét több konstans értékkel kell összehasonlítani. Az utasítás általános alakja:

```

switch ( kifejezés )
{
    case konstans_kifejezés :
        utasítások
    case konstans_kifejezés :
        utasítások
    default :
        utasítások
}

```

A **switch** utasítás először kiértékeli a *kifejezést*, majd átadja a vezérlést arra a **case** (eset) címke-re, amelyben a *konstans_kifejezés* értéke megegyezik a kiértékelt *kifejezés* értékével, majd a program futása ettől a ponttól folytatódik. Amennyiben egyik **case** konstans sem egyezik meg a *kifejezés* értékével, a program futása a **default** címkevel megjelölt utasítástól folytatódik. Ha nem használunk **default** címket, akkor a vezérlés a **switch** utasítás blokkját záró kapcsos zárójel (/) után utasításra adódik.

Amikor a **case** vagy a **default** címkevel belépünk a **switch** utasítás törzsébe, akkor attól a ponttól kezdve a megadott utasítások sorban végrehajthatódnak (a blokkok záró-jel eléréséig). Általában azonban az adott esethez tartozó programrészlet végrehajtása után a **goto**, a **break** vagy a **return** utasítással kilépünk a **switch** utasításból. Ha a **switch** utasítás után álló utasításokkal kívánjuk folytatni a program futását, akkor a **break** utasítást használjuk.

A **switch** használatára tipikus példa az előző alfejezet kalkulátor programjának átirat-
valtozata:

```

#include <iostream>
using namespace std;

const char PROMPT = " : ";

```

```

#include <iostream>
using namespace std;

void main()
{
    cout<<"A valasz [I/N]?"<<endl;
    char valasz=cin.get();

    switch (valasz) {
        case 'I': case 'i':
            cout<<"A valasz IGEN."<<endl;
            break;
    }
}

```

A következő példában azt mutatjuk be, hogyan lehet több esethez ugyanazt a programrészletet rendelni. A programban a válaszként beolvasott karaktert feldolgozó **switch** utasításban az 'i' és 'I', illetve az 'n' és 'N' esetekhez tartozó **case** címkeket egymás után helyeztük el.

A **return** utasítással, mind a **switch** utasításból, mind pedig a **main()** függvényből ki lépünk. Altalában a **default** címke utáni utasításokat is **break**-kel zárjuk, azon egyszeresrű oknál fogva, hogy a **default** bárhol elhelyezkedhet a **switch** utasításon belül.

```

void main()
{
    double a,b,e;
    char op;
    cout.put(PROMPT);
    cin>>a>>b;
    cin.get();
    switch (op) {
        case '+':
            e = a + b;
            break;
        case '-':
            e = a - b;
            break;
        case '*':
            e = a * b;
            break;
        case '/':
            e = a / b;
            break;
        default:
            cerr<<"Hibas muvelet!"<<endl;
            cin.get();
    }
    cout<<a<<op<<b<<endl;
    return;
}

```



```

case 'n':
    cout<<"A valasz NEM."<<endl;
    break;
default:
    cout<<"Hibas valasz!"<<endl;
    break;
}

```

6.4 A ciklusutasítások

A programozási nyelveken bizonyos utasítások automatikus ismétlését biztosító programszervezetet *iterációnak* vagy *ciklusnak* (*loop*) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul. A C++ nyelv háromféle ciklusutasítást tartalmaz, melyek formája:

```

while (kifejezés) utasítás

for (kifejezés1opt : kifejezés2opt : kifejezés3opt) utasítás

do utasítás while (kifejezés)

```

A **for** utasítás esetén az *opt* index arra utal, hogy a megjelölt kifejezések használata opcionális.

A ciklusokat csoportosíthatjuk a vezérlőfeltétel kiértékelésének helye alapján. Azokat a ciklusokat, amelyeknél az *utasítás* végrehajtása előtt kerül feldolgozásra a vezérlőfeltétel, *előillesztő* ciklusnak nevezzük. Ezeknél a ciklus soron következő iterációja csak akkor hajtodik végre, ha a feltétel igaz (nem nulla). A **while** és a **for** előillesztő ciklusok.

Ezzel szemben a **do** ciklus legalább egyszer mindig lefut, hisz a vezérlőfeltétel ellenőrzése az *utasítás* végrehajtása után történik – *hátnillesztő* ciklus.

Mindhárom ciklusfajta esetén a helyesen szervezett ciklus befejezi a működését, amikor a vezérlőfeltétel hamissá (nulla) válik. Vannak esetek azonban, amikor számdékosan vagy véletlenül olyan ciklust hozunk létre, melynek vezérlőfeltétele soha sem lesz hamis. Ezeket a ciklusokat *végtelen ciklusnak* nevezzük:

```

for ( ;; ) utasítás;

while (1) utasítás;

do utasítás while (true);

```

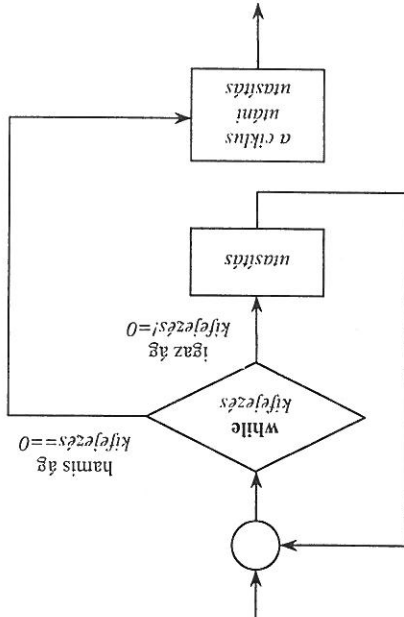
A ciklusokból a vezérlőfeltétel hamis értékének bekövetkezése előtt is ki lehet ugrani (a végtelen ciklusból is). Erre a célra további utasításokat biztosít a C++ nyelv, mint a **break**, a **return**, illetve a ciklus törzsen kívültre irányuló **goto**. A ciklus törzsének bizonyos utasításait átugorhatjuk a **continue** utasítás felhasználásával. A **continue** hatására a ciklus következő iterációjával folytatódik a program futása.

6.4.1 A while ciklus

A **while** ciklus mindaddig ismétli a hozzá tartozó *utastíást* (a ciklus törzset), amíg a vizsgált *kifejezés* (vezérlőfeltétel) értéke igaz (nem nulla). A vizsgálat mindig megelözi az *utastíás* végrehajtását. A **while** jól olvasható alakja:

while (*kifejezés*)
utastíás

A **while** ciklus működésének folyamata az alábbi ábrán követhető nyomon.



A következő példaprogramban meghatározzuk az első n egész szám összegét:

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

void main()
{
    long sum;
    int n = 2003;

    cout<<"Az első "<<n<<" egész szám "

```

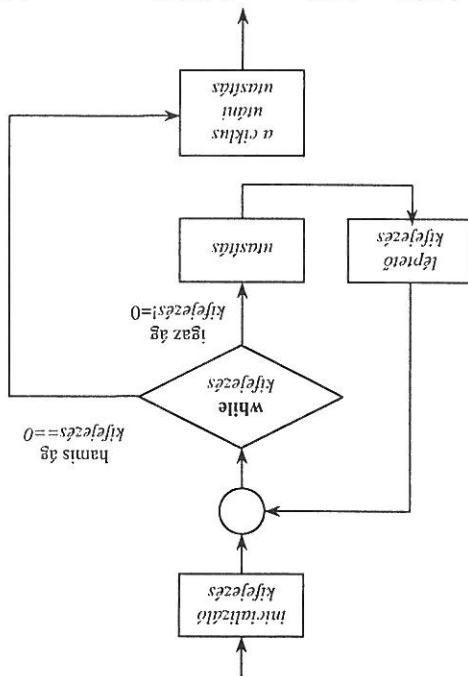
```

while (n>0) { // vagy : // while (sum += n--,n) ;
    sum += n; // while (n>0)
    n--; // sum += n--;
}
cout<<"összege: "<<sum<<endl;
cin.get();
}

```

6.4.2 A for ciklus

A **for** utasítást általában akkor használjuk, ha a ciklusmagban megadott utasítást adott számúszor kívánjuk végrehajtani. A **for** ciklus működési vázlatát az alábbi ábra tartalmazza:



A **for** utasítás általános alakjában külön megjelöljük az egyes kifejezések szerepét:

```
for (init_kif ; feltetel_kif ; lepteto_kif) utasítás
```

A **for** utasítás valójában a **while** utasítás speciális alkalmazása, így a fenti **for** ciklus minden további nélkül átirható **while** ciklussá:

```

init_kif;
while (feltetel_kif) {
    utasítás;
    lepteto_kif;
}

```

Megjegyezzük azonban, hogy a **while** és a **for** ciklusok eltérően viselkednek, ha a ciklus törésében a később ismertetésre kerülő **continue** utasítást használjuk. Az ábrán jól nyomonkövethető a **for** ciklus végrehajtásának lépései:

1. Megtörténik az *init_kif* kifejezés (amennyiben megadtuk) kiértékelése, melynek során a ciklusban használt változókat inicializáljuk. Semmilyen megkötés nincs az *init_kif* típusára vonatkozóan.
2. A következő lépésben a *feltevel_kif* (aritmetikai vagy mutató típusú) kifejezést dolgozza fel a rendszer (ha megadtuk). A *feltevel_kif* függvényében a ciklus az alábbi három eset valamelyikének megfelelően működhet:
 - Ha a *feltevel_kif* értéke igaz (nem nulla, **true**), akkor végrehajtódik az utasítás melyet a *lepteto_kif* kifejezés kiértékelése követ. Minden további iterációt a *feltevel_kif* kiértékelése nyit meg, és a *lepteto_kif* kiértékelése zár.
 - Ha a *feltevel_kif* nincs megadva, akkor annak értékét igaznak tételezi fel a rendszer, és a ciklus futása pontosan megegyezik az előző esetbenél tárgyalattal. Ebben az esetben a ciklus leállítása csak a **break**, a **return** vagy a **goto** utasítások segítségével oldható meg.
 - Ha a *feltevel_kif* értéke hamis (0, **false**), akkor a **for** utasítás befejezi működését, és a vezérlés a program következő utasítására kerül.

Példaként a **for** ciklusra, írjuk át a **while** ciklussal megoldott, egész számok összegét meghatározó programot! Azonnal látható, hogy a megoldásnak ez a változata sokkal áttekinthetőbb és egyszerűbb:

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

void main()
{
    long sum;
    int i, n = 2003;
    cout<<"Az első "<n<<" egész szám ";
    for (i=1, sum=0; i<=n; i++)
        sum += i;
    cout<<"összege: "<<sum<<endl;
    cin.get();
}
```

A példában szereplő ciklus magjában csak egyetlen kifejezés-utasítás található, ezért az alábbi tömörítési lépések elvégezhetők:

```
for (i=1, sum=0; i<=n; sum += i, i++) ;

//vagy

for (i=1, sum=0; i<=n; sum += i);
```

A ciklusokat egymásba is ágyazhatjuk, hisz a ciklus *utasítás* újabb ciklusutasítás is lehet. A következő példában kettős ciklust használunk a megadott méretű szorozótábla megjelenítésére:

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

const int MAX = 8;

void main()
{
    int i, j;

    for (j=1; j<=MAX; j++)
        cout<<"\t"<<j;
    cout<<endl;

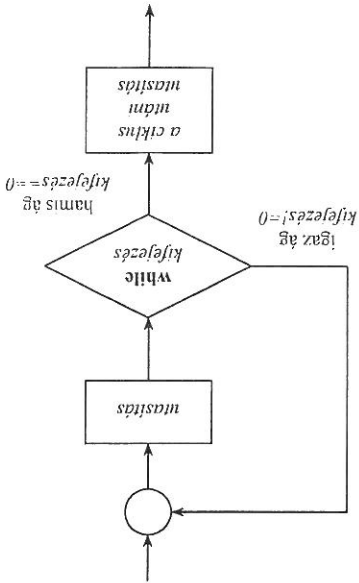
    for (i=1; i<=MAX; i++)
    {
        cout<<i;
        for (j=1; j<=MAX; j++)
            cout<<"\t"<<i*j;
        cout<<endl;
    }
}
```

6.4.3 A do-while ciklus

Mint ahogy a ciklusok bevezető részében említettük, a **do-while** utasításban a ciklus törzset képező utasítás végrehajtása után kerül sor a tesztelésre. Így a ciklus törzse legalább egyszer mindig végrehajtodik. A **do-while** utasítás használatának jól olvasható formája:

```
do
    utasítás
while (kifejezés);
```

A **do-while** ciklus futása során mindig az *utasítás* végrehajtását követi a *kifejezés* kiértékelése. Ha a *kifejezés* értéke igaz (*nem 0*, *true*), akkor új iteráció kezdődik, míg hamis (*0*, *false*) érték esetén a ciklus befejezi működését. A **do-while** ciklus működését a mellékelt ábrán blokkdiagram segítségével szemléltethetjük:



Első példaként készítsük el az egész számokat összegző programnak a **do-while** ciklust használó változatát! Gyakorlatilag semmit sem kell megváltoztatnunk, a ciklus átszervezésén kívül:

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

void main() {
    long sum;
    int n = 2003;
    cout<<"Az első "<<n<<" egész szám ";
    sum=0;
    do {
        sum += n;
        //
        sum += n--;
        //
        while (n>0);
        //
        cout<<"összege: "<<sum<<endl;
    }
    cin.get();
}
```

A második példánk egy játékpogram, amelyet futtatva a felhasználónak 0 és 100 közötti számot kell kitálmnia. A program **do-while** ciklusa egy **else-if** szerkezetet is tartalmaz, amely a felhasználó választát értékei.

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

void main() {
    int szam, tipp, lepes = 0;
    // a véletlenszám-generátor inicializálása
    srand(unsigned(ctime(NULL)));
    szam = rand() % 101;
    cout<<"Gondoltam egy számot 0 és 100 között... "<<endl;
    do {
        cout<<"Tipp: ";
        cin>>tipp;
        cin.get();
        if (tipp == szam)
            cout<<"\nKitalált, a gondolt szám: "<<szam<<endl;
        else if (tipp > szam)
            cout<<"Nem talált, túl nagy!"<<endl;
        else
            cout<<"Nem talált, túl kicsi!"<<endl;
        lepes++;
    } while (tipp != szam);
    cout<<"Gratullok, "<<lepes<<" lépésben sikerült!"<<endl;
    cin.get();
}
```

A példában a 0 és 100 közé eső gondolt szám előállítását a „pszeudodovéletlen” számot szolgáltató *rand()* függvénnnyel végezzük:

```
szam = rand() % 101;
```

Ahhoz azonban, hogy minden futáskor új véletlen szám keletkezzen, a véletlenszámgenerátort véletlenszerűen (az idővel) inicializáljuk a

```
srand( unsigned( time( NULL ) ) );
```

hívás segítségével.

6.5 A break és a continue utasítások

Vannak esetek, amikor egy ciklus szokásos működésébe közvetlenül be kell avatkoznunk. Ilyen feladat például, amikor adott feltétellel teljesülése esetén ki kell ugrani a ciklusból, vagy amikor a ciklus végrehajtását a következő iterációval kívánjuk folytatni. Ezen feladatok elvégzésére a legtöbb programozási nyelv a *goto* utasítás használatára hagyatkozik, azonban a C++ nyelvén külön utasítások – a *break* és a *continue* – állnak a programozó rendelkezésére.

6.5.1 A break utasítás

A *break* hatására az utasítást tartalmazó legközelebbi *switch*, *while*, *for* és *do-while* utasítások működése megszakad, és a vezérlés a megszakított művelet utáni első utasításra kerül. A *break switch* utasításban történő felhasználását már bemutattuk 6.3. fejezetben, most csak a ciklusból való kigúráásra mutatunk példákat.

Az alábbi ciklus akkor lép ki, ha megtalálja a legnagyobb olyan 2003-nál kisebb egész számot, amely 123-mal osztható:

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

void main() {
    int n=2003;
    while (n>0) {
        if (n % 123)
            break;
        n--;
    }
    if (n)
        cout<<n<<endl;
    cin.get();
}
```


kiértékelésével kezdődik. A **for** ciklus esetében azonban, a feltétellel kifejezés kiértékelését megelőzi a lépítő kifejezés feldolgozása.

A következő példában a **continue** segítségével értük el, hogy a 0-tól 99-ig egyesével lépkedő ciklusban csak a 7-tel vagy 11-gyel osztható számok jelenjenek meg:

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

void main()
{
    int i;
    for (i=0; i<100; i++) {
        if (i % 7 && i % 11)
            continue;
        cout<<i<<endl;
    }
    cin.get();
}
```

A **break** és a **continue** utasítások gyakori használata rossz programozói gyakorlatot jelent. Erdemes mindig átgondolnunk, hogy nem lehet-e az adott programszerkezetet **break**, illetve **continue** nélkül megvalósítani.

Az alábbi ciklus, amely az *<Enter>* lenyomásáig olvas, és ír ki karaktereket

```
while ( (ch = getch()) != EOF)
{
    if (ch == '\n')
        break;
    putchar(ch);
}
```

minden további nélkül átirható a **break** felhasználása nélkül:

```
while ( (ch = getch()) != EOF && ch != '\n')
    putchar(ch);
```

Hasonló a helyzet a **continue** utasítással. A következő ciklus a *<Ctrl+Z>* és *<Enter>* billentyűk lenyomásáig minden beírt karaktert visszatér a képernyőre, kivéve a sormentes karaktert:

```
while ( (ch = getch()) != EOF)
{
    if (ch == '\n')
        continue;
    putchar(ch);
}
```

A fenti működést a **continue** utasítás nélkül megvalósító ciklus:

```
while ( (ch = getchar()) != EOF)
    if (ch != '\n')
        putchar(ch);
```

6.6 A goto utasítás

A strukturált, jól áttekinthető (így valószínűleg hibátlan) programszerkezet kialakítása során nem szabad **goto** utasítást használnunk. A **goto** utasítás ugyanis kuszává, áttekinthetetlené teszi a forrásprogramot. Vannak esetek azonban, amikor a **goto** segítségével jutunk el legegyszerűbben a megoldáshoz.

A **goto** utasítás felhasználásához utasításcímkevel kell megjelölnünk azt az utasítást, ahova később ugrani szeretnénk. Az utasításcímke valójában egy azonosító, amelyet kettősponttal határolnunk el az utána álló utasítástól:

```
azonosító: utasítás
```

A **goto** utasítás, amellyel a fenti címmel megjelölt sorra adhatjuk a vezérlést:

```
goto azonosító;
```

Szükséges megjegyeznünk, hogy a **goto** utasításnak és a célcímkeknek egyazon függvényen belül kell elhelyezkednie.

Nézzünk egy olyan példát, ahol a **goto** használatával nélkül igen bonyolult programszerkezet áll elő! A feladat az, hogy két egymásba ágyazott ciklusból lépünk ki, ha egy bizonyos feltétel bekövetkezik. Az egyszerű változatban a **goto** utasítást használjuk:

```
for ( ... ) {
    for ( ... ) {
        if ( ... )
            goto stop;
    }
}
```

```
stop:
    cout<<"Hiba van a programban!"<<endl;
```

Amennyiben **goto** nélkül kívánjuk megoldani a fenti feladatot, segédváltozót kell bevezetnünk:

```

kesz = false;
for ( ... ) {
    for ( ... ) {
        if ( ... ) {
            kesz = true;
            break;
        }
        . . .
    }
    if (kesz) break;
}
cout<<"Hiba van a programban!"<<endl;

```

6.7 A return utasítás

A **return** utasítás befejezi az éppen futó függvény működését, és a vezérlés visszakérül a hívó függvényhez. Ha a *main()* függvényben használjuk a **return** utasítást, akkor programunk befejezi a futását, hisz a *main()* függvény az („öt hívó”) operációs rendszernek adja vissza a vezérlést.

A **return** utasítást

return ;

alakban használva olyan függvényből léphetünk ki, amely a nevével nem ad vissza semmilyen értéket (**void** típusú függvény, eljárást). A függvények többsége azonban valamilyen értékkel (a függvényértékkel) tér vissza a hívás helyére, amit a **return** utasításban adhatunk meg:

return kifejezés ;

A **return** utasítással részletesen a függvényeket tárgyaló fejezetben foglalkozunk.

6.8 Kivételek kezelése

Kivételeknek (*exception*) nevezzük azt a hibás állapotot vagy eseményt, amely megszakítja az alkalmazás szabályszerű futását. A kivételkezelés lehetővé teszi, hogy a C++ program futása során a vezérlés és az információ automatikusan ahhoz a programrészhez kerüljön, amelyik „hajlandó” az adott típusú kivétel kezelésére. A **throw** utasítás segítségével tetszőleges típusú kivételt (*exception*) továbbíthatunk („dobhatunk”) a kezelőhöz (*exception handler*), amelyik azt elkapva (**catch**), elvégzi az esemény feldolgozását.

A C++ nyelv a kivételek kezelését a megszakításos (*termination*) modell alapján végzi. Ez azt jelenti, hogy a kivételt kiváltó programrészlet (függvény) futása megszakad az esemény bekövetkeztekor. A kivételkezelést általában hibakezelés, illetve hibátűró (*fault-tolerant*) programok kialakításakor alkalmazzuk.

A C++ nyelv típusorientált kivételkezelései lehetőségeit az alábbi három elem jellemzi:

- kivételkezelés alatt álló programrészlet kijelölése (**try**-blokk),
- a kivétel továbbbírása (**throw**),
- a kivétel „elfogása” és kezelése (**catch**).

A szabványos C++ nyelvben viszonylag kevés beépített kivétel érhető el, azonban a különböző osztálykönyvtárak előszeretettel alkalmazzák a kivételeket a hibás állapotok jelzésére. Az *exception* fejláblomány tartalmazza az *exception* osztály leírását, mely a különböző logikai és futás-idejű kivételek alapszálya.

A kivételkezelést a programunk adott részén belül lokálisan kell kialakítanunk a **try**, a **throw** és a **catch** kulcsszavak felhasználásával. A kivételkezelés csak a **try**-blokknak (próbálkozás-blokknak) nevezett utasításban megadott (illetve az abból hívott) kód végrehajtása esetén fejti ki hatását. A kijelölt kódrészleten belül a **throw** utasítással

throw kifejezés;

adhatjuk át a vezérlést a kifejezés típusának megfelelő kezelőnek (*handler*), amelyet a **catch** kulcsszó követően adunk meg.

```
try // kivételfigyelés alatt álló utasítások
{
    utasítások;
}
catch (kivétel_1-deklaráció) // a kivétel_1 kezelője
{
    utasítások;
}
catch (kivétel_2-deklaráció) // a kivétel_2 kezelője
{
    utasítások;
}
```

Függetlenül eseten a függvény fejlécben megadhatjuk, hogy mely kivételek továbbí-
tódjanak a függvényen kívüli kezelőhöz:

```
// minden kivétel továbbíttódik
int f1();

// csak char* típusú kivételek továbbíttódnak
int f2() throw(char *);

// egyetlen kivétel sem továbbíttódik
int f3() throw();
```

A beérkező kivételek a típusuk alapján a megfelelő kezelőhöz irányítódnak:

```

catch (char *s) // a char* kivétel kezelője
{
    // a kezelő törzse
}
catch (...) // minden más kivétel kezelője
{
    // a kezelő törzse
}

```

Amikor egy kivételt a **throw** utasítással továbbítunk, akkor az utasításban megadott kifejezés értéke átmásolódik a **catch** utasítással kijelölt kezelő paraméterébe, így a kezelőben lehetőség nyílik ezen érték feldolgozására.

Amikor egy olyan kivételt továbbítunk, amelynek nincs kezelője, a futtató rendszer a **terminate()** függvényt aktivizálja, amely kilépteti a programunkat (**abort()**). Ha egy olyan kivételt továbbítunk, amelyik nincs benne az adott függvény által továbbítandó kivételek listájában, akkor az **unexpected()** rendszerhívás állítja le a programunkat. Mindkét kiléptési folyamatba beavatkozhatunk saját kezelők definíciójával, melyek regisztrációját az *except* fejlécben deklarált **set_terminate()**, illetve **set_unexpected()** függvényekkel végezhetjük el.

Az alábbi példában a számológép programunkat felkészítettük a kivételes események (kilépés, hiba adatbevitel stb.) kezelésére. A megoldásban függvényeket használtunk, melyekkel a 8. fejezetben ismerkedünk meg részletesen.

```

#include<iostream>
using namespace std;

void beolvas(double &a, double &b, char &op)
    throw (bool, const char*) {
    char p[100];
    cout<<" ";
    // biztonságos adatbevitel
    cin.getline(p,100);
    int db=sscanf(p,"%lf%lf", &a, &b);
    if (toupper(p[0])=='X') // kilépés
        throw true;
    if (db!=3)
        throw "Hiba: adatbevitel.";
}

```

```

double szamol(double a, double b, char op)
{
    throw (int, const char *) [
        double e=0;
        switch (op) {
            case '+': e=a + b;
            case '-': e=a - b;
            case '*': e=a * b;
            case '^': e=pow(a,b);
            case '/': if (b)
                break;
            else
                throw 1;
            e=a / b;
            break;
        }
        default: throw "Hibás operátor";
    }
    return e;
}

void main()
{
    double x,y;
    char op;
    while (true) {
        try {
            beolvas(x,y,op);
            cout<<'='<<szamol(x,y,op)<<endl;
        }
        catch (bool) {
            cout<<"Visszlat!"<<endl;
            break; // while
        }
        catch (int) {
            cout<<"Szamolasi hiba."<<endl;
        }
        catch (const char *s) {
            cout<<s<<endl;
        }
        catch (...) {
            cout<<"Ismeretlen kivetel"<<endl;
        }
    }
    cin.get();
}

```

6.9 Definíciók bevitel az utasításokba

A C nyelvben a változók deklarációját (definícióját) a blokkok elején az utasítások előtt kell elhelyeznünk. A C++ megengedi, hogy a változók deklarációját bárhova helyezzük a program kódján belül. Egyetlen feltétel, hogy a változót mindeenképpen deklarálnunk (definíciálnunk) kell a felhasználása előtt. Elve ezzel a lehetőséggel a változó deklarációját (definícióját) és a felhasználása közél helyezhető, elkerülve ezzel bizonyos gyakori programozási hibákat.

```
// Változók deklarációja az első felhasználás közélében
#include <iostream>
using namespace std;
```

```
void main()
{
    cout << "Kerek egy számot: ";
    int n;
    cin >> n;
    for (int i=1; i<=n; i++)
        cout<<i<<endl;
}
```

A példában a **for** ciklusba helyeztük a ciklusváltozó definícióját. Bizonyos esetekben a változó-definíciót feltételi használó utasításokba is bevihetjük, amennyiben a változó azonnal inicializáljuk, például véletlen számmal:

```
if (int s = rand()%12) {
    cout<<s<<endl;
}
while (int n = rand()%23) {
    cout<<n<<endl;
}
```

Felhívjuk a figyelmet arra, hogy az utasításokban definiált változók csak az utasításon belül használhatók. Így ha a ciklusváltozó értékére cikluson kívül is szükségünk van, nem szabad ezt a megoldást alkalmaznunk.

7. Származtatott adattípusok

Az eddigi példákban olyan változókat használtunk, amelyek csak egyetlen érték (*skalár*) tárolására voltak alkalmasak. A programozás során azonban gyakran van arra szükség, hogy azonos vagy különböző típusú elemekből álló adattárhalmat a memóriában tároljunk, és az adatokon valamilyen műveleteket hajtsunk végre. C++ nyelvben a tömbök, illetve a felhasználói típusok (**struct**, **class**, **union**) segítségével elegendősen megoldhatjuk ezeket a problémákat.

7.1 Tömbök, sztringek és mutatók

A tömb (*array*) azonos típusú adatok halmaza, amelyek a memóriában folytonosan helyezkednek el. Az elemek elérése a tömb nevét követő indexelés operátorban megadott elemsorszám (index) segítségével történik. A tömb tehát olyan tárolók készlete, melyekre közös névvel és egy indexszel hivatkozunk.

A leggyakrabban használt tömbtípus egyetlen kiterjedéssel (dimenzióval) rendelkezik. Az egydimenziós tömböket vektornak is szokás nevezni. Azonban ún. többdimenziós tömbök használatára is adott a lehetőség. Kétdimenziós tömbök esetén az elemek tárolása soronként (sorfolytonosan) történik.

7.1.1 Egydimenziós tömbök

Az egydimenziós tömböket definiálnunk kell, melynek általános alakja:

```
típus tömbnév[méret];
```

A definícióban szereplő *típus*, amely az elemek típusát definiálja, a **void** és a függvénytípus kivételével tetszőleges típus lehet. A szögletes zárójelek között a tömb méretét adjuk meg. Ez a *méret*, amelynek a fordító által kiszámítható konstans kifejezésnek kell lennie, a tömbben tárolható elemek számát definiálja.

Alátalánosan is elmondható, hogy *T* típus esetén a *[Tméret]* egydimenziós tömb (vektor) *méret* darab, *T* típusú elem tárolására alkalmas. Az elemeket *0*-tól (*méret-1*)-ig indexeljük.

Példaként tekintsünk egy 5 elemet tartalmazó egész tömböt, melynek elemeit az indexek négyzetével töltjük fel! A tömb definíciója:

```
int a[5];
```

A tömb elemeinek egymás után történő elérésére általában a **for** ciklust használjuk, melynek változója a tömb indexe. A tömb elemeire pedig az indexelés operátorának ((/)) segítségével hivatkozunk.

```
for (int i = 0; i < 5; i++)  
    a[i] = i * i;
```

Az alábbi ábrán felvázoltuk a tömb elhelyezkedését a memóriában. Minden elem esetén felüntetjük az elem értékét a ciklus lefutása után, és az elemre való hivatkozás módját.

a	a
a[4]	16
a[3]	9
a[2]	4
a[1]	1
a[0]	0
a tömb neve	

A tömb számára a memóriában lefoglalt terület mérete a *sizeof(a)* kifejezéssel pontosan lekérdezhető, míg a *sizeof(a[0])* kifejezés egyetlen elem méretét adja meg. Így a két kifejezés hányadosából (egész osztás) mindig megtudható a tömb elemeinek száma:

```
int a[10];  
int n = sizeof(a) / sizeof(a[0]);
```

Felhívjuk a figyelmet arra, hogy a C++ nyelv semmilyen ellenőrzést nem végez a tömb indexeire vonatkozóan. Az indexhatár átlépése a legkülönbözőbb hibákhoz vezethet, melyek felderítése sok időt vesz igénybe.

```
double nap[23];  
nap[-1] = 12.23; // hiba!  
nap[23] = 356.23; // hiba!
```

Az indexhatárok átlépése általában nem közvetlen módon történik, hanem például a kezelő ciklus rossz beállításával. Sajnos ez a hiba csak ritkán vezet jól érzékelhető hibáuzemeléshez, ehelyett legtöbbször csak a változónk misztikus megváltozásából következtethetünk a hibára.

Az alábbi példában a 4-elemű vektorba beolvasott számokat átlagoljuk, majd kirtjuk az átlagot és az egyes elemek elérését ettől az átlagtól. Gyakori megoldás, hogy a tömb méretét makró vagy konstans segítségével adjuk meg. Így, ha a méretet később meg kell változtatnunk, akkor csak egyetlen helyen kell a programot módosítanunk.

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

const int NUM = 4;

void main()
{
    double szam[NUM];
    double atlag = 0.0;
    int i;
```

```
    for (i = 0; i < NUM; i++) // az elemek beolvasása
    {
        cout<<"szam["<<i<<" ] = ";
        cin>>szam[i];
        atlag += szam[i]; // az elemek összegzése
    }
    cin.get();
    atlag /= NUM; // az átlag kiszámítása
    cout<<endl<<"Az atlag: "<<atalg<<endl;
```

A program futási eredményeinek tanulmányozásakor felhívjuk a figyelmet az adatbevitel megvalósítására:

szam[0] = 1.2		
szam[1] = 0.9		
szam[2] = 2.3		
szam[3] = 7.9		
Az atlag: 3.075		
0.	1.2	1.875
1.	0.9	2.175
2.	2.3	0.775
3.	7.9	-4.825

7.1.1.1 Az egydimenziós tömbök inicializálása

A C++ nyelv lehetővé teszi, hogy a tömböket a definíciójuk során inicializáljuk. Ez a kezdőértékkadás eltér az egyszerű változók esetén használt megoldástól:

```
tipus tömbnev[méret] = { vesszővel tagolt inicializációs lista };

Nézzünk néhány példát vektorok inicializálására!

int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

```
char szo[8] = { 'a', 'l', 'm', 'a' };
float szamok[] = { 12.3, 23.4, 34.5, 45.6 };
```

A második példában az inicializációs lista kevesebb értéket tartalmaz, mint a tömb elemeinek száma. Ekkor a szó tömb első 4 eleme felveszi a megadott értékeket, míg a többi elem értéke a tárolási osztálytól függően vagy 0 (**extern, static**) vagy pedig határozatlan (**auto**) lesz.

Az utolsó példában a *szamok* tömb elemeinek számát az inicializációs listában megadott konstansok száma (4) megfelelően állítja be a fordítóprogram. Jól használható ez a megoldás, ha a tömb elemeit fordításonként változtatjuk. A kérdés csak az, hogyan tudjuk meg a programon belül a tömb méretét? A választ az elemszám előzőekben bemutatott meghatározása adja:

```
float szamok[] = { 12.3, 23.4, 34.5, 45.6 };
```

```
#define NSZAM (sizeof(szamok) / sizeof(szamok[0]))
```

vagy

```
const int nszam = sizeof(szamok) / sizeof(szamok[0]);
```

A blokkon belül létrehozott (**auto**) tömbök esetén az inicializációs lista tetszőleges futásidejű kifejezést tartalmazhat:

```
double a[3] = { sqrt(2), exp(1), sin(3.14159265/3) };
```

7.1.1.2 Egydimenziós tömbök és a *typedef*

Mint már említettük a programunk olvashatóságát növeli, ha a bonyolultabb típusneveket szinonim nevekkel helyettesítjük. Erre származtatott típusok esetén is a **typedef** biztosít lehetőséget.

Legyen a feladatunk két 8-elemű egész vektor összege egy harmadik vektorban! Az összegzés után az eredményvektort másoljuk át egy negyedik vektorba! A feladat megoldásához szükséges tömböket kétféleképpen is létrehozhatjuk:

```
int a[8], b[8], c[8], d[8];
```

vagy

```
typedef int vektor8[8];
vektor8 a, b, c, d;
```

A C++ nyelv az indexelésen kívül semmilyen más műveletet sem definiál a tömbökre vonatkozóan. Ezért az összegezést és az elemek átmásolását (tömb-tömb közötti értékadás) saját magunknak kell elvégezni.

A példaprogramban az *a* és *b* vektorokat konstansokkal inicializált vektorként definiáltuk:

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

typedef int vektor8[8];

vektor8 a = {1, 2, 3, 4, 5, 0, 1, 3};
vektor8 b = {4, 3, 2, 1, 0, 5, 4, 2};

void main()
{
    vektor8 c, d;
    int i;

    for (i = 0; i < 8; i++) // az elemek összegzése
        c[i] = a[i] + b[i];

    for (i = 0; i < 8; i++) // a c vektor másolása d-be
        d[i] = c[i];

    cout<<"nd = ";
    for (i = 0; i < 8; i++)
        cout<<d[i]<<" ";
    cout<<endl;
    cin.get();
}
```

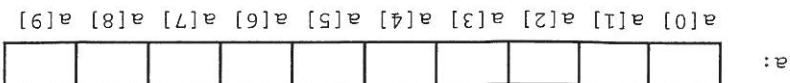
7.1.2 Mutatók és a tömbök

A C++ nyelvben a mutatók és a tömbök között szoros a kapcsolat. Minden művelet, ami tömbindexeléssel elvégezhető, mutatók segítségével szintén megvalósítható. Az egydimenziós tömbök (vektorok) és az egyszerűsített mutatók között 100%-os (tartalmi és formai) analógia áll fenn. A többdimenziós tömbök és a többszörös indirektességű mutatók között ez a kapcsolat azonban csak formai.

Nézzük meg, honnan származik ez a vektorok és az egyszerűsített mutatók között fennálló szoros kapcsolat! Defináljunk egy 10 elemű egész vektort!

```
int a[10];
```

A vektor elemei a memóriában adott címtől kezdve folytonosan helyezkednek el. Mindegyik elemre *a[i]* formában hivatkozhatunk:

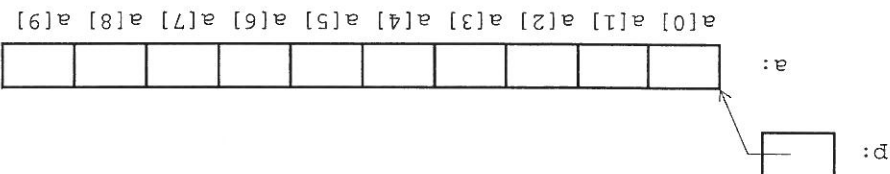


Most vegyünk fel egy p egészeze mutató pointert, majd a „címe” operátor segítségével állítsuk az a tömb elejére (a 0. elemre)!

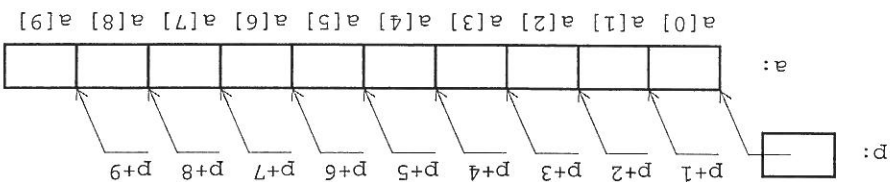
```
int *p;
```

```
p = &a[0];
```

Ezek után, ha hivatkozunk a p mutató által kijelölt ($*$) objektumra (változóra), akkor valójában az $a[0]$ elemet érjük el.



Ha p változóra mutat, akkor a mutatóaritmetika szabályai alapján a $p+1$, a $p+2$ stb. címek az adott elem után elhelyezkedő elemeket jelölik ki. (Megjegyezzük, hogy negatív számokkal a változót megelőző elemeket címezhetjük meg.) Ennek alapján a $*(p+i)$ kifejezéssel a tömb minden elemét elérhetjük:



A p mutató szerepe teljesen megegyezik az a tömbnév szerepével, hisz mindkettő az elemek sorozatának kezdetét jelöli ki a memóriában. Lényeges különbség azonban a két mutató között, hogy míg a p mutató változó (tehát értéke tetszőlegesen módosítható), addig az a egy konstans mutató, amelyet a fordító rögzít a memóriában.

Ezek után általánosíthatunk tetszőleges típusú a tömbre és ugyanolyan típusú p mutatóra. Ha a mutatót az alábbi módszerek valamelyikével a tömb első elemére irányítjuk,

```
p = &a[0];
```

vagy

```
p = a;
```

akkor azonosak a következő, egy sorban elhelyezkedő hivatkozások:

```
char s2[] = "NATPA";
```

```
char s1[10] = "NATPA";
```

stringliterálok (stringkonstansek) felhasználásával:

A karaktertömbök inicializálása azonban sokkal biztonságosabban elvégezhető a

tert megadtunk az inicializációs listában.

Az *s1* string számára 10 bajt helyet foglal a fordító, és az első 5 bajtba bemásolja a megadott karaktereket. Az *s2* azonban pontosan annyi bajt hosszú lesz, ahány karakter-

```
char s2[] = { 'N', 'A', 'T', 'P', '\0' };
```

```
char s1[10] = { 'N', 'A', 'T', 'P', '\0' };
```

nem szabad megfellelkezni a `\0` karakter elhelyezéséről:

A programozás során gyakran használunk kezdőértékekkel ellátott stringeket. A kezdő-
érték megadására alkalmazhatjuk a vektorként bemutatott megoldásokat, azonban

```
char sor[81];
```

Amikor helyet foglalunk valamely karaktersorozat számára, akkor a string végét jel-
ző bajtot is figyelembe kell vennünk. Ha az *str* tömbben maximálisan 80 karakteres
szöveget szeretnénk tárolni, akkor a tömb méretét $80+1=81$ -nek kell megadnunk:

'C'	'+'	'+'	' '	'n'	'y'	'e'	'l'	'v'	'\0'
-----	-----	-----	-----	-----	-----	-----	-----	-----	------

(*char*), amelyben a karakter sorozat végét *nulla* értékű bajt (0) jelzi:

Az egydimenziós tömböket leggyakrabban karakter sorozatok tárolására használjuk. A
C++ nyelv definíciója nem rendelkezik önálló stringtípussal, ezért karaktertöm-
bökben valósítja meg a stringek tárolását. A string tehát olyan karaktertöm-
b

7.1.3 Karakter sorozatok (stringek)

A C++ fordító az *a[i]* hivatkozásokat automatikusan $*(a+i)$ alakúra alakítja, majd ezt
a pointeres alakot lefordítja. Az analógia azonban visszafelé is igaz, vagyis az indi-
rektesség (*) operátora helyett mindig használhatjuk az indexelés ([]) operátort.

- A tömb i-dik eleme: $a[i]$ $p[i]$ $*(a+i)$ $*(p+i)$
- A tömb 0-dik eleme: $a[0]$ $p[0]$ $*p$ $*(a+0)$ $*(p+0)$
- A tömb i-dik elemének címe: $\&a[i]$ $\&p[i]$ $p+i$

A sztringek kezelésére karaktermutatókat is használhatunk, azonban a mutatókkal óvatosan kell bánni. Tekintsük az alábbi gyakran alkalmazott definíciókat! Első esetben a fordító létrehozza a 16-elemű s tömböt, majd oda bemásolja az inicializáló szöveg karaktereit, valamint a 0-as karaktert. A második esetben a fordító az inicializáló szöveget elárólja a sztringliterálok számára fenntartott területen, majd a sztring kezdőcímevel inicializálja a létrejövő *ps* mutatót.

```
char s[16] = "alfa";
```

```
char *ps = "gamma";
```

A *ps* értéke a későbbiek folyamán természetesen megváltoztatható (ami a jelen példában a "gamma" sztring elvesztését okozza):

```
ps = "iota";
```

Ekkor valójában mutató-értékadás történik, hisz a *ps* felveszi az új sztringkonstans címet. Ezért az s tömb nevére irányuló értékadás fordítási hibához vezet:

```
s = "iota"; // hibás!
```

A C++ nyelv a sztringekre vonatkozóan szintén nem tartalmaz semmilyen műveletet (értékadás, összehasonlítás stb.). Azonban a karaktertorsorozatok kezelésére szolgáló könyvtári függvények sokkal több lehetőséget biztosítanak a programozónak, mint más nyelvek sztringműveletei. Nézzünk néhány sztringekre vonatkozó alapművelet elvégzésére szolgáló függvényt!

<i>Művelet</i>	<i>Függvény</i>
szöveg beolvasása	<i>scanf()</i> , <i>gets()</i>
szöveg kírása	<i>cin>></i> , <i>cin.getline()</i>
Ertékadás	<i>strcpy()</i>
Hozzáfűzés	<i>strcat()</i>
sztring hosszának lekérdézése	<i>strlen()</i>
sztringek összehasonlítása	<i>strcmp()</i>

(Sztringkezelő függvények használata esetén a *cstring* deklarációs állományt be kell építenünk a forrásprogramba.)

Amennyiben egy sztringen karakterenként kell végiglépkedni, akkor választhatunk a tömbös és a pointeres megközelítés között. A következő példaprogramban a beolvasott karaktertorsorozatot először títkosítjuk a kizáró vagy művelet felhasználásával, majd

visszaállítjuk az eredeti tartalmat. (A titkosításhoz a karaktertömb, míg a visszakódolásnál a mutató értelmezést használjuk.)

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

const unsigned int KULCS=0xE7;
```

```
void main()
{
    char s[80], *p;
    cout<<"Kerek egy szoveget" : " ";
    cin.getline(s,80);

    for (int i = 0; s[i] != '\0'; i++) // titkositas
        s[i] ^= KULCS;

    cout<<"A titkosított szoveg : "<<s<<endl;

    p=s;
    while (*p)
        *p++ = KULCS;
    cout<<"Az eredeti szoveg : "<<s<<endl;
    cin.get();
}
```

Mind a két esetben a ciklusok leállási feltétele a sztringet záró nullás bajt elérése volt.

7.1.3.1 A C++ könyvtár string típusa

A C++ nyelv szabványos sablonkönyvtárában (STL) a szövegek kezelést támogató osztályokat is találunk. Az előzőekben bemutatott megoldások a C-stílusú karaktersorozatokra vonatkoznak, most azonban megismerkedünk a *string* típus lehetőségeivel. (A későbbiekben mindkét sztringtípust használjuk a példaprogramokban).

A C++ stílusú karakterisorozat-kezelés eléréséhez a *string* nevű fejállományt kell a programunkba beépíteni:

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

Ezt követően a *string* típuskal definiált objektumokon keresztül egy sor kényelmes szöveggel művelet áll a rendelkezésünkre, operátorok és tagfüggvények formájában. Nézzünk néhányat ezek közül! (A táblázatban a tagfüggvények előtt pont szerepel. A tagfüggvények nevét az objektum neve után, ponttal elválasztva adjuk meg.)

<i>Művelet</i>	<i>C++ megoldás</i>
szöveg beolvasása	<code>cin>>, getline()</code>
szöveg kiírása	<code>cout<<</code>
értékkadás	<code>=, assign()</code>
összetűzés	<code>+, +=</code>
string hosszának lekérdezése	<code>.size()</code>
stringek összehasonlítása	<code>.compare()</code>
átalakítás C stílusú karakter sorozattá	<code>.c_str(), .data(), .begin()</code>

Irjuk át szövegitkösítő programunkat C++ stílusú stringkezelés felhasználásával!

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

void main()
{
    string s;
    char *p;
    cout<<"Kerek egy szöveget : ";
    getline(cin, s);
    for (int i = 0; s[i]; i++)
        // titkosítás
        s[i] ^= KULCS;
    cout<<"A titkosított szöveg : "<<s<<endl;

    p=(char *)s.c_str();
    while (*p)
        *p++ ^= KULCS;
    cout<<"Az eredeti szöveg : "<<s<<endl;
    cin.get();
}
```

7.1.4 Többdimenziós tömbök

A C++ nyelv támogatja a többdimenziós tömbök használatát. A többdimenziós tömbök deklarációjának általános formája:

```
tipus tömbnév[méret1][méret2]...[méretn];
```

ahol dimenzióként kell megmondani a méreteket. A gyakorlatban a többdimenziós tömbök helyett általában mutatótömböket alkalmazunk.

A fejezetben csak a kétdimenziós tömbök bemutatására és használatára szorítkozunk, azonban a kétdimenziós tömbök (mátrixok) esetén alkalmazott megoldások több dimenzióra is általánosíthatók.

Tároljuk az alábbi 3x5-ös, egész elemeket tartalmazó mátrixot, a C++ program megfelelő adatstruktúrájában!

$$\begin{bmatrix} 4 & 26 & 90 & 19 & 11 \\ 13 & 30 & 70 & 79 & 9 \\ 7 & 87 & 60 & 23 & 12 \end{bmatrix}$$

A definícióban kezdőértékként megadhatjuk a mátrix elemeit, csak arra kell ügyelnünk, hogy sortolytonos legyen a megadás.

```
matrix[3][5] = { { 4, 26, 90, 19, 11 },
                  { 13, 30, 70, 79, 9 },
                  { 7, 87, 60, 23, 12 } } ;
```

A tömb elemeinek eléréséhez az indexelés operátort használjuk méghozzá kétszer. A

```
matrix[2][3]
```

hivatkozással a 2. sor 3. sorszámú elemét (23) jelöljük ki. Nem szabad megfeledkez-nünk arról, hogy az indexek értéke minden dimenzióban 0-val kezdődik!

Az alábbi ábrán a kétdimenziós *matrix* tömb elemei mellett felüntetjük a sorok és az oszlopok (s/o) indexeit is.

<i>A sorok címe:</i>		s/o	0	1	2	3	4
matrix[0]	→	0	4	26	90	19	11
matrix[1]	→	1	13	30	70	79	9
matrix[2]	→	2	7	87	60	23	12

A táblázat első oszlopában található kifejezések, amelyeket a második dimenzió elhagyásával kapunk, a tömb sorainak kezdőcímét szolgáltatják. Így érthetővé válik a megállapítás, hogy a kétdimenziós tömb valójában egy olyan vektor (egydimenziós tömb), melynek elemei vektorok (mutatók). Ennek ellenére a többdimenziós tömbök mindig folytonos memóriaterületen helyezkednek el. A példánkban a mátrix soraí képezik azokat a vektorokat, amelyekből a *matrix* vektor felépül.

Használva a vektorok és a mutatók közötti formai analógiát, az indexelés operátorai minden további nélkül átirthatók indirektitásig operátorává. Az alábbi kifejezések a kétdimenziós tömb ugyanazon elemére hivatkoznak:

```
matrix[1][2] * (matrix[1] + 2) * (*(matrix+1)+2)
```

A következő programrészelben először megkeressük a *matrix* legnagyobb (*emax*), illetve legkisebb (*emin*) elemét:

```
int emax, emin;
emax=emin=matrix[0][0];
for (int i = 0; i < 3; i++)
for (int j = 0; j < 5; j++) {
    if (matrix[i][j] > emax)
        emax = matrix[i][j];
    if (matrix[i][j] < emin)
        emin = matrix[i][j];
}
```

Majd mátrixos alakban megjelentjük a *matrix* nevű kétdimenziós tömb tartalmát:

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 5; j++)
        cout<<"\t"<<matrix[i][j] <<endl;
}
```

7.1.5 Mutatótömbök, stringtömbök

A C++ programok többsége tartalmaz olyan szövegeket, például üzeneteket, amelyeket adott index (hibakód) alapján kívánunk kiválasztani. Az ilyen szövegek tárolására a leggyorsabb megoldás a stringtömbök használata.

A stringtömbök kialakítása során választhatunk a kétdimenziós tömb és a mutatótömb között. Kezdő C++ programozók számára sokszor gondot jelent ezek megkülönböztetése.

Tekintsük az alábbi két definíciót!

```
int a[5][10];
int *b[5];
```

Formailag az *a[2][5]* és a *b[2][5]* hivatkozások egyaránt helyesek, hiszen mindkét esetben egyetlen *int* típusú elemet jelölnek. Azonban az *a* egy „igazi” (statikus) kétdimenziós tömb, amely számára a fordító 50 darab *int* típusú elem tárolására alkalmas folytonos területet foglal le a memóriában. Az elemek helyének meghatározására a fordító az alábbi hagyományos kifejezést használja:

```
10 * sor + oszlop
```

Ezzel szemben a *b* 5-elemű mutatóvektor. A fordító csak az 5 darab mutató számára foglal helyet a definíció hatására. A inicializáció további részeit a programból kell elvégeznünk. Inicializáljuk úgy a mutatótömböt, hogy az alkalmas legyen 5x10 egész elem tárolására!

```
int s1[10], s2[10], s3[10], s4[10], s5[10];
int *b[5] = { s1, s2, s3, s4, s5 };
```

Látható, hogy az 50 **int** elem tárolására szükséges memóriaterületen felül, további területet is felhasználunk (a mutatók számára). Joggal vetődik fel a kérdés, hogy mi az előnye a mutatótömbök használatának. A választ a sorok hosszában kell keresni. Míg a kétdimenziós tömb esetén minden sor ugyanannyi elemet tartalmaz,

addig a mutatótömb esetén az egyes sorok mérete tetszőleges lehet. A alábbi definíció-nak megfelelő struktúrát szintén ábrázoltuk.

```
static int s1[1], s2[3], s3[10], s4[6], s5[8];
int *b[5] = { s1, s2, s3, s4, s5 };
```

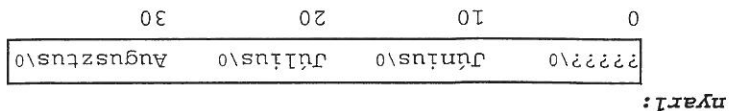
→	b[4]									
→	b[3]									
→	b[2]									
→	b[1]									
→	b[0]									

A mutatótömb másik előnye, hogy a felépítése összhangban van a dinamikus memóriafoglalás lehetőségeivel, így fontos szerepet játszik a dinamikus helyfoglalású tömbök kialakításánál.

A stringtömböket általában kezdőértékek megadásával definiáljuk. Nézzünk példát stringtömbök kialakítására kétdimenziós karaktertömb, illetve karakter típusú mutatókat tartalmazó pointertömb felhasználásával!

```
char myarr1[10] = { "?????",
                    "Júnus",
                    "Július",
                    "Augusztus" };
```

Az első definíció során egy 4x10-es karaktertömb jön létre, amelyben a sorok számát a fordítóprogram az inicializációs lista alapján határozza meg. A kétdimenziós karaktertömb sorai a memóriában folytonosan helyezkednek el:

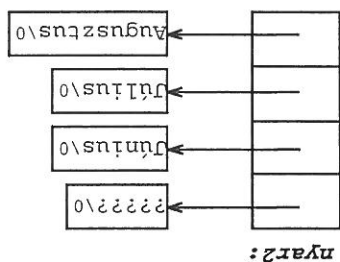


A második esetben mutatótömböt használunk a nyári hónapok neveinek tárolására. Eredmes összehasonlítani a két megoldást mind a definíció, mind pedig a memóriaelérés szempontjából. A sztringtömb definíciója,

```
char *nyar2[] = { "????",
                  "jún\0",
                  "júl\0",
                  "aug\0" };

```

amellyel a memóriában négy különálló területet foglaltunk le, melyeket teljesen feltölti a fordító:



Mindkét esetben az első index megadásával a sztringekre hivatkozunk, míg mindkét index használatával a kiválasztott sztring adott karakterét érjük el.

```
cout<<"Honap : "<<nyar1[3]<<endl;
cout<<"Honap : "<<nyar2[3]<<endl;
cout<<"A honap első betűje : "<<nyar1[3][0]<<endl;
cout<<"A honap első betűje : "<<nyar2[3][0]<<endl;

```

A képernyőn az alábbi sorok jelennek meg az utasítások végrehajtása után:

```
Honap : Augusztus
Honap : Augusztus
A honap első betűje: A
A honap első betűje: A

```

7.1.6 Dinamikus helyfoglalású tömbök

Tömböket használó programok fejlesztése során nagyon hamar memóriakorlátokba ütközhetünk. Ezért a C++ programban a nagyobb tömbök létrehozását és felszabadítását nem bízuk a fordítóra, hanem a dinamikus memóriakezelés lehetőségeit kihasználva magunk gondoskodunk e műveletek elvégzéséről. Mint már említettük, a dinamikus memóriahasználat alap gondolata, hogy az adataink számára csak akkor foglalunk helyet, amikor szükségünk van rájuk. Amennyiben az adatok feleslegessé válnak, az általuk elfoglalt memóriaterületet azonnal felszabadítjuk.

C++ nyelven a dinamikus memóriakezeléshez mutatókat használunk. A tömbök és mutatók közötti (tartalmi és formai) analógia tisztázása után nincs akadálya annak, hogy tömbök számára dinamikusan foglaljunk memóriaterületet. A memóriafoglalás elvégzéséhez a **new** operátort, míg a felszabadításához a **delete[]** operátort használjuk (lásd az 5.14. alfejezetet).

A dinamikus tömbök alkalmazási lehetőségeinek részletezése helyett három példa-programot mutatunk, melyek jól szemléltetik az elmondottakat. A feladat neveket beolvasása és névsorba rendezése. Mindhárom megoldásban sztringtömböt használunk a nevek tárolására, amit különböző módon hozunk létre.

Megoldás statikus kétdimenziós karaktertömb segítségével:

A megoldáshoz használt `nev[20][51]` tömb felépítése:

20.	
19.	
...	
4.	Júlia
3.	Adrienn
2.	Natalia
1.	Anna
0.	Léna

A 20 sor, soronként 51 karaktert tartalmazó tömböt statikusan hozzuk létre. Emiatt a nevek maximális száma korlátozott, így azt ellenőriznünk kell az adatbevitel során. A megoldásban aláhúzással kiemeljük a lényeges programsorokat.

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

int main() {
    const int maxn=20;
    char nev[maxn][51];
    int db;
```

```

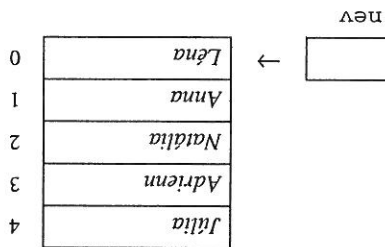
// A nevek darabszámainak bekérése
cout<<"Hány nevet kíván rendezni: ";
cin>>db; cin.get();
if (db<1) return -1;

new= new tsor[db];
// dinamikus memóriafoglalás

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

int main() {
    typedef char tsor[51];
    tsor *nev;
    int db;
}

```



A sorok száma tetszőleges, azonban a nevek legfeljebb 50 karakteresek lehetnek. Valójában egydimenziós dinamikus tömböt használunk, azonban a tömbelemek maguk is egydimenziós karaktertömbök.

Megoldás egydimenziós, 51-karakteres szöveges elemeket tartalmazó dinamikus tömbbel:

```

// A nevek darabszámainak bekérése
cout<<"Hány nevet kíván rendezni: ";
cin>>db; cin.get();
if (db>maxn || db<1) return -1;

// kilépés

for (int i=0; i<db; i++) {
    cout<<"<i>i<<"<". nev: ";
    cin.getline(nev[i], 51);
}

char sv[51];
for (int i=0; i<db-1; i++)
    for (int j=i+1; j<db; j++)
        if (strcmp(nev[i], nev[j])>0) {
            strcpy(sv, nev[i]);
            strcpy(nev[i], nev[j]);
            strcpy(nev[j], sv);
        }
    for (int i=0; i<db; i++)
        cout<<nev[i]<<endl;
}

// kilírás

```



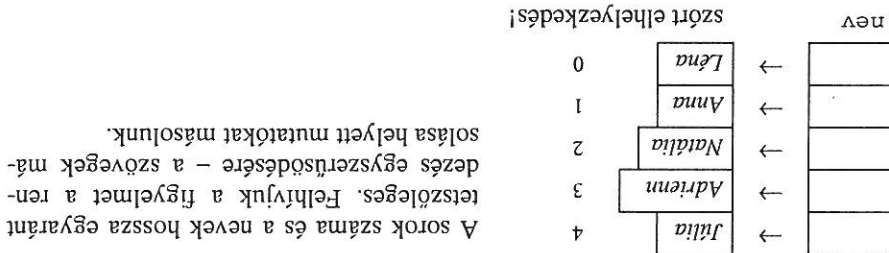
```

// Dinamikus memóriafoglalás a mutatótömb számára
nev=new string[db];

// A nevek darabszámaának bekérése
cout<<"Hány nevet kíván rendezni: ";
cin>>db; cin.get();
if (db<1) return -1;

int db;
string *nev;
typedef char * string;
int main() {
    using namespace std;
    #include <iostream>
    #include <cstring>

```



Megoldás dinamikus mutatótömb felhasználásával:

```

}
    cin.get();
delete [] nev;
// A lefoglalt memória felszabadítása

for (int i=0; i<db; i++) // kiírás
    cout<<nev[i]<<endl;

delete [] sv;
}
strcpy(nev[j],sv);
strcpy(nev[i],nev[j]);
strcpy(sv,nev[i]);
if (strcmp(nev[i],nev[j])>0) {
    for (int j=i+1; j<db; j++)
        for (int i=0; i<db-1; i++)
            char * sv=new char[51];
// rendezés
}
    cin.getline(nev[i],51);
for (int i=0; i<db; i++) {
    cout<<i<<" nev: ";
    // a nevek beolvasása

```

```

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

int main()
{
    string *nev;
    int db;

```

Végezetül nézzük meg a feladatnak egy igazi C++-os megoldását, ahol az utolsó változat adatstruktúrához hasonló struktúrát használunk, azonban alkalmazzuk a szabványos sablonkönyvtár lehetőségeit! A sorok száma és a nevek hossza egyaránt tetszőleges lehet.

Megoldás dinamikus objektumtömb felhasználásával:

```

}
cin.get();

delete [] nev;
// A mutatótömb felszabadítása

delete [] nev[];
for (int i=0; i<db; i++)
// A nevek számára lefoglalt memóriaterület felszabadítása
cout<<nev[i]<<endl;
for (int i=0; i<db; i++)
// Kilírás

}
nev[j]=p;
nev[i]=nev[j];
char * p=nev[i];
if (strcmp(nev[i],nev[j])>0) {
for (int j=i+1; j<db; j++)
for (int i=0; i<db-1; i++)
// Rendezés a mutatók felcserélésével

}
strcpy(nev[i],sv);
// A név másolása
nev[i]=new char[strlen(sv)+1];
// Helyfoglalás a név számára
cin.getline(sv,251);
cout<<i<<" . nev: ";
for (int i=0; i<db; i++) {
char sv[251];
// A nevek beolvasása (mérhető az inputpuffer mérete)

```

```
// A nevek darabszámaának bekérése
cout<<"Hány nevet kíván rendezni: ";
cin>>db; cin.get();
if (db<1) return -1;
```

```
// Dinamikus memóriafoglalás az objektumtömb számára
new= new string[db];
```

```
// A nevek beolvasása
for (int i=0; i<db; i++) {
    cout<<"<i>".név: ";
    getline(cin,nev[i]);
}
```

```
// Rendezés
for (int i=0; i<db-1; i++)
    for (int j=i+1; j<db; j++)
        if (nev[i].compare(nev[j])>0)
            swap(nev[i],nev[j]);
```

```
// Kijírás
for (int i=0; i<db; i++)
    cout<<nev[i]<<endl;
```

```
// Az objektumtömb felszabadítása
delete [] nev;
cin.get();
}
```

A programunk még tovább egyszerűsödik a *sort()* algoritmus felhasználásával. Az *algoritmus* fejállomány beépítése után a rendezést végző programrészt egyetlen utasítással alakítható:

```
// Rendezés
sort( &nev[0], &nev[db] );
```

7.2 Felhasználó által definiált adattípusok

A C++ nyelv megengedi, hogy a nyelv meglévő típusait felhasználva újabb típusokat hozzunk létre. Az eddigiek folyamán már többször élünk ezzel a lehetőséggel, amikor a *typedef* segítségével szinonim típusneveket vezetünk be. Ugyancsak ide tartoznak a felsorolt (*enum*) típusok, amelyeket csoportos, egymással kapcsolatban álló, konstansok létrehozására használunk. Az *enum* típusnevönömagában semmi sem jó, hiszen a felsorolt típust a C++ nyelv felhasználójának (a programozónak) kell definiálnia, az alábbi formában:

```
enum válasz { nem, igen } ;
```

Ebben a fejezetben a struktúra, az osztály, a bitmező és az unió típusokkal foglalkozunk. A tömb- és a felhasználói típusokat közös néven összeállított (*aggregate*) típusoknak nevezzük. A fenti típusok közül az ismerkedést a struktúrával (**struct**) kezdjük. A struktúrátípusokkal kapcsolatos fogalmak és megoldások minden további nélkül alkalmazhatók az osztály, a bitmező és az unió típusokra is.

7.2.1 A *struct* struktúrátípus

A programozás során gyakran találkozhatunk olyan problémákkal, amelyek megoldásához különböző típusú adatokat önálló programozási egységben kell feldolgozunk. Tipikus területe az ilyen jellegű feladatoknak az adatbázis-kezelés, ahol a fájl tárolási egysége (a rekord) tetszőleges mezőkből épülhet fel.

C++ nyelven a struktúra (**struct**) típus több tetszőleges típusú (kivéve a **void** és a függvénytípus) adatok együttese. Ezek az adatelemek, melyek szokásos elnevezése struktúraelem vagy adattag (*data member*), önálló, a struktúrán belül érvenyes nevekkel rendelkeznek. (A más nyelveken megszokott mező (*field*) elnevezést a később ismertésre kerülő bitstruktúrák esetén használja a C++ nyelv.)

A struktúrátípusú változó létrehozása két lépésben végezhető el. Először deklarálunk kell magát a struktúrátípust, melyet felhasználva változókat definiálhatunk. A struktúra szerkezetét meghatározó deklaráció általános formája:

```
struct struktúrátípus {
    típus1 tag1; // kezdőérték nélkül!
    típus2 tag2;
    ...
    típusN tagN;
};
```

Felhívjuk a figyelmet arra, hogy a struktúra deklarációja azon kevés esetek egyike, ahol a kapcsolatos zárójel mögé a pontosvesszőt kötelező kitenni. Az adattagok deklarációjára a C++ nyelv szokásos deklarációs szabályai érvényesek. A fenti típusú változót a már megismert módon készíthetünk:

```
struct struktúrátípus struktúra_változó; // C/C++
struktúrátípus struktúra_változó; // C++
```

A C++ nyelven a **struct**, **union** és **class** kulcsszavak után álló név típusnévként használható a kulcsszó megadása nélkül.

Nézzünk egy konkrét példát a struktúrátípus megadásáról: Könnyításképző program készítése során jól alkalmazható a következő adatstruktúra:

```

struct book {
    char nev [20] ; // a szerző neve
    char cím [40] ; // a mű címe
    int év; // a kiadás éve
    float ár; // a könyv ára
};

```

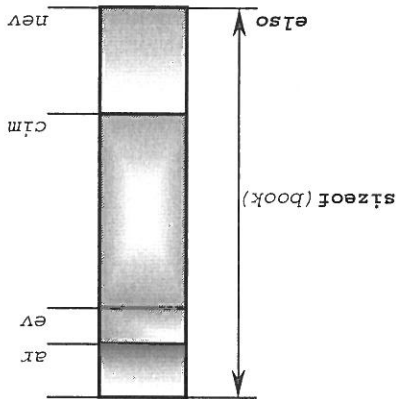
A típusdeklaráció után változókat is készíthetünk:

```
book macska, gyerek, cprog;
```

A struktúra definíciójával létrehozunk egy új felhasználói típust. A struktúrátípusú változó adatait a fordító a deklaráció sorrendjében tárolja a memóriában. Az alábbi ábrán grafikusan ábrázoltuk a

```
book elso;
```

definícióval létrehozott adatastruktúra felépítését.



Az ábráról is leolvasható, hogy az adatagok nevei a struktúra elejétől mért távolságot jelölnek. A struktúra mérete általában megegyezik az adatagok méretének összegével. Bizonyos esetekben azonban (optimalizálás sebessége, adatok memóriahatárát való igazítása stb.) „lyukak” keletkezhetnek a struktúra tagjai között. A **sizeof** operátor alkalmazásával azonban mindig a pontos méretet kapjuk.

7.2.1.1 Hivatkozás a struktúra adattagjaira

A struktúra szót gyakran önállóan is használjuk, ilyenkor azonban nem a típusra, hanem az adott struktúrátípussal létrehozott változóra gondolunk. Definíciójunk néhány változót az előzőekben deklarált *book* típus felhasználásával!

```
book s1, s2, *ps ;
```

Az *s1* és az *s2 book* típusú, statikus helyfoglalású változók, amelyek tárolásához szükségességes memóriaterület lefoglalásáról a fordító gondoskodik. A *ps* egy pointer, amely *book* típusú tárolóra mutathat. Ahhoz, hogy a *ps* mutatóval is hivatkozhassunk a struktúra, két lehetőség közül választhatunk. Az első, kevésbé hasznos esetben a *ps-t* egy-szerűen ráíranyítjuk az *s1* struktúrára:

```
ps = &s1;
```

A második lehetőség a dinamikus memóriafoglalás alkalmazását jelenti. Az alábbi programrészelben memóriát foglalunk a *book* struktúra számára, majd pedig felszabadítjuk azt:

```
ps = new (nothrow) book
if (!ps) exit(-1);
// ...
delete ps;
```

A megfelelő definíciók megadása után három struktúrával rendelkezünk: *s1*, *s2* és *ps*. Nézzük meg, hogyan lehet értéket adni a struktúráknak! Erre a célra a C++ nyelvben a pont (.) operátor használható:

```
strcpy(s1.nev, "Bjorne Stroustrup");
strcpy(s1.cim, "The C++ Programming Language");
s1.ev = 2002;
s1.ar = 23.12;
```

A pont operátor bal oldali operandusa a struktúráváltatózó, a jobb oldali operandusa pedig a struktúrán belül jelöli ki az adattagot.

Amikor a pont operátort a *ps* által mutatott struktúrára alkalmazzuk, a precedencia-szabályok miatt zárójelek között kell megadnunk a *ps* kifejezést:

```
strcpy((ps).nev, "Bjorne Stroustrup");
strcpy((ps).cim, "The C++ Programming Language");
(ps).ev = 2002;
(ps).ar = 23.12;
```

Mivel a C++ nyelvben (főleg a C nyelv első változataiban) gyakran használunk mutatókkal kijelölt struktúrákat (például függvények argumentumaként), a nyelv ezekben az esetekben egy önálló operátort – a nyíl (>) operátor – biztosít az adattag-hivatkozások elvégzésére. (A nyíl operátor két karakterből, a mínusz és a nagyobb jelből áll.) A nyíl operátor használataival olvashatóbb formában írhatjuk fel *ps* által kijelölt struktúra adattagjaitra vonatkozó értékadásokat:

```
strcpy(ps->nev, "Bjorne Stroustrup");
strcpy(ps->cim, "The C++ Programming Language");
ps->ev = 2002;
ps->ar = 23.12;
```

A nyíl operátor bal oldali operandusa a struktúráváltózóra mutató pointer, míg a jobb oldali operandusa – a pont operátorhoz hasonlóan – a struktúrán belül jelöli ki az adat-tagot. Ennek megfelelően a *ps->ar* kifejezés jelentése: "a *ps* mutató által kijelölt struktúra ar adattagja".

A „cime” (& operátor az *s1* struktúra alkalmazva szintén mutatóhoz jutunk, amellyel már használható a nyíl operátor:

```
(&s1)->ev=2002;
```

Láthatjuk, hogy a pont és a nyíl operátorok mindkét esetben – közvetlen és a közvetett hivatkozás esetén – egyaránt használhatók. Szem előtt tartva a program olvashatóságát és helyességet, javasoljuk, hogy a pont operátort csak közvetlen (a struktúrátípusú változó adattagjára történő) hivatkozás esetén, míg a nyíl operátort kizárólag közvetett (mutató által kijelölt struktúra adattagjára vonatkozó) hivatkozás esetén használjuk.

A struktúra vonatkozó értékadás speciális esete, amikor egy struktúrátípusú változó tartalmát egy másik, azonos típusú változónak kívánjuk megfeleltetni. Ezt a műveletet adattagokonként is elvégezhetjük,

```
strcpy(s2.nev, s1.nev);
strcpy(s2.cim, s1.cim);
s2.ev = s1.ev;
s2.ar = s1.ar;
```

azonban a C++ szabvány értelmezi a struktúráváltózókra vonatkozó értékadás (=) műveletét is:

```
s2 = s1; // Ez megfelel meg a fenti 4 értékadásnak
*ps = s2;
s1 = *ps = s2;
```

Az értékadásnak ez a módja valóban egyszerűen a struktúra által lefoglalt memória-blokk átmásolását jelenti. Az értékadás művelete azonban gondot okoz, amikor a struktúra olyan mutatót tartalmaz, amellyel külső memóriablokkra hivatkozunk:

```
struct string {
    char *p;
    int len;
} s1, s2;

s1.p = "Hello";
s2 = s1;
```

A másolás végzetével a külső memóriablokk (string) mindkét struktúrához hozzátartozik, hiszen csak a mutatók tartalma (a blokk címe) másolódott át. A fentiekhez hasonló problémák megoldására több lehetőség közül is választhatunk. Egyrészt az adattagonkénti értékadás módszerét használva magunknak kell kiküszöbölni a problémát, másrészt pedig a másolás műveletének átdefiniálásával (*operator overloading*) saját értékadó operátort hozhatunk létre a struktúrához.

Az alábbi példában billentyűzetről töltjük fel a *book* típusú struktúrát, majd megjelöljük a tárolt adatokat:

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

struct book {
    char nev[20];
    char cim[40];
    int ev;
    float ar;
};

void main() {
    book wb;

    // az adatok beolvasása
    cout<<"\nKérem a könyv adatait!\n";
    cout<<" Szerzo      : "; cin.getline(wb.nev,20);
    cout<<" Cim        : "; cin.getline(wb.cim,40);
    cout<<" Kiadas eve : "; cin>>wb.ev;
    cout<<" Ar (Ft)   : "; cin>>wb.ar;
    cin.get();

    // az adatok megjelenítése
    cout<<"\nA könyv adatai : \n";
    cout<<" Szerzo      : "<<wb.nev<<endl;
    cout<<" Cim        : "<<wb.cim<<endl;
    cout<<" Kiadas eve : "<<wb.ev<<endl;
    cout<<" Ar (Ft)   : "<<wb.ar<<endl;
    cin.get();
}
```

7.2.1.2 Kezdőértékadás a struktúrában

A tömbökhöz hasonlóan a struktúra definíciójában is szerepelhet kezdőértékadás. Az egyes adattagokat inicializáló kifejezések vesszővel elválasztott listáját kapcsos zárójel közé kell zárni. Példaként lássuk el kezdőértékkel *book* típusú *s* struktúrát!

```
book s1 = {"Bjarne Stroustrup", "The C++ Programming Language",
2002, 23.12 };
```


Amennyiben a struktúra valamely adattagja tömb, akkor a kezdőértékek listájában a tömb inicializálását végző részt külön kapcsos zárójelek között adjuk meg. A zárójelek használata nem kötelező, de az inicializálás biztonságosabbá tehető vele:

```
struct Vektor {
    int nem;
    int v[20];
};

Vektor a = { 5, { 1, 2, 3, 4, 5 } };
Vektor b = { 4, 10, 20, 30, 40 } ;
```

7.2.1.3 Egyéb ágyazott struktúrák

Már említettük, hogy a struktúráknak tetszőleges típusú adattagjai lehetnek. Ha egy struktúrában valamilyen más struktúrátípusú adattagot használunk, ún. egymásba ágyazott struktúrát kapunk.

Tételezzük fel, hogy síkbeli geometriai objektumok adatait struktúra felhasználásával kívánjuk feldolgozni! Az alábbi struktúra a geometriai alakzat helyét meghatározó pont koordinátáinak tárolására alkalmas:

```
struct pont {
    int x;
    int y;
};
```

A kör definiáló struktúrában a kör középpontját az előzőleg deklarált *pont* adattagban tároljuk:

```
struct kor {
    pont kp;
    int r;
};
```

Hozzunk létre két kört, még hozzá úgy, hogy az egyiknél használjunk kezdőérték-adást, míg a másikat adattagonkénti értékadással inicializáljuk!

```
kor k1 = { { 100, 100 }, 50 }, k2;

k2.kp.x = 50;
k2.kp.y = 150;
k2.r = 200;
```

A kezdőértékkadás során a belső struktúrát inicializáló konstansokat szintén nem kötelező kapcsolos zárójelek közé helyezni. A *k2* struktúra adatait *kp* struktúra értékadás során az első pont (.) operátorral a *k2*-ben elhelyezkedő *kp* struktúra hivatkozunk, majd ezt követi a belső struktúra adatait *kp* struktúra hivatkozása.

Ha a *pont* struktúrát máshol nem használjuk, akkor névtelen struktúráként közvetlenül beírhatjuk a *kör* struktúrába:

```
struct kör {
    int x;
    int y;
    } kp;
    int r;
};
```

Bonyolultabb dinamikus adatszerkezetek (például lineáris lista) kialakításánál adott típusú elemeket kell láncba fűznünk. Az ilyen elemek általában valamilyen adatot és egy mutatót tartalmaznak. A C++ nyelv lehetővé teszi, hogy a mutatót az éppen deklarálás alatt álló struktúrátípusával definiáljuk. Az ilyen struktúrákat, amelyek önmagukra mutató pontintert tartalmaznak adattagként, önhivatkozó struktúráknak nevezzük. Példaként tekintsük az alábbi *listaelem* deklarációt!

```
struct listaelem {
    int adattag;
    listaelem * kapcsolat;
};
```

Ez a rekurzív deklaráció mindössze annyit tesz, hogy a *kapcsolat* mutatóval az adott struktúra mutathatunk. A fenti megoldás nem ágyazza egymásba a két struktúrát, hiszen az a struktúra, amelyre a későbbiek során a mutatóval hivatkozunk, valahol máshol fog elhelyezkedni a memóriában.

A C++ fordító számára a deklaráció elsősorban azért szükséges, hogy a deklarációnak megfelelően tudjon memóriát foglalni, vagyis hogy ismerje a létrehozandó objektum méretét. A fenti deklarációban a létrehozandó tároló egy mutató, amelynek mérete független a struktúra méretétől.

7.2.1.4 Struktúrák

Már láttunk példát arra, hogyan lehet struktúrában tömbtípusú adattagot elhelyezni. Most azonban azt nézzük meg, hogy milyen módon lehet struktúraelemeket tartalmazó tömböket használni! Struktúrából pontosan ugyanúgy kell definiálni, mint bármilyen más típusú tömböt. Példaként az előzőekben deklarált *book* típust használva hozzunk létre egy 100-kötetes „könyvtár”!

```
book lib[100];
```

A kérdés már csak az, hogyan tudunk hivatkozni a tömbelemstruktúrák adataigjaira. Ebben az esetben a pont és az indexelés operátorát együtt kell használnunk:

```
lib[23].ar = 12.23;
```

Mivel két operátornak azonos a precedenciája, a kiértékelés során a balról-jobbra szabályt használja a fordító. Tehát először a tömbelem kerüli kijelölésre (az indexelés), amit az adattagra való hivatkozás (pont operátor) követ. Így zárójelek használata nem szükséges, hiszen a fenti kifejezés az alábbi kifejezéssel egyenértékű:

```
(lib[23]).ar = 12.23;
```

A struktúrátömböt a definiálásakor a szokásos módon inicializálhatjuk. A áttekinthetőség érdekében ajánlott az egyes struktúrák kezdőértékét kapcsolós zárójelben elkülöníteni:

```
book lib[] = { { "0. Iro", "0. Konyv", 2003, 1000 },
                { "1. Iro", "1. Konyv", 2004, 2000 },
                { "2. Iro", "2. Konyv", 2005, 3000 } };
```

Amennyiben dinamikusan kívánjuk a „könyvtár” létrehozni, mutatót kell használnunk az azonosításra:

```
book * lib;
```

A struktúraelemek számára a **new** operátorral foglalthatunk helyet a dinamikusan kezelt memóriaterületen:

```
lib = new book[100];
```

A tömbelemben tárolt struktúra a pont operátor segítségével hivatkozhatunk:

```
lib[23].ar = 12.23;
```

Ha már nincs szükségünk a struktúra elemeire, akkor a **delete[]** operátorral felszabadítjuk a lefoglalt memóriaterületet:

```
delete[] lib;
```

Bizonyos könyvtárműveletek (például rendezés) hatékonnyabban elvégezhetők, ha műtatómbben tároljuk a dinamikusan létrehozott könyvek mutatóit:

```
book * lib[100];
```

A struktúraelemek számára az alábbi ciklus segítségével foglalthatunk helyet a dinamikusan kezelt memóriaterületen:

Ekkor a tömbelem által kijelölt struktúrára a nyíl operátor segítségével hivatkozhatunk:

```
for (int i=0; i<100; i++)
    p1ib[i]=new book;
```

```
p1ib[23] -> ar = 12.23;
```

Ha már nincs szükségünk a struktúra elemeire, akkor az egyes elemeken végighaladva felszabadítjuk a lefoglalt memóriaterületet:

```
for (int i = 0; i < 100; i++)
    delete p1ib[i];
```

Az alábbi példában a dinamikusan létrehozott, 100 könyvet tartalmazó könyvtárból, kigyűjtjük az 1979 és 2005 között megjelent műveket. (A programnak az a része, amely a könyvtár véletlen adatokkal való feltöltését végzi, a későbbiek során fájlikézési műveletekkel helyettesíthető.)

```
#include <iostream>
#include <string>
#include <cstdlib>
#include <ctime>
#include <cstdlib>

using namespace std;

struct book {
    char nev[20];
    char cim[40];
    int ev;
    float ar;
};

const int KSZAM = 100;

int main()
{
    int i;
    // Helyfoglalás ellenőrzéssel
    bool * talalat = new bool[KSZAM];
    book * d1ib = new (nothrow) book[KSZAM];
    if (!d1ib) {
        cerr<<"\a\Nincs elég memória!\n";
        return -1;
    }
}
```

A C++ nyelv az objektum-orientált programépítés megvalósításához egyrészt kibővíti a C nyelv `struct` típusát, másrésztől pedig új `class` típust vezet be. Mindkét típus alkalmas arra, hogy osztályt definiáljunk segítségével. (Az osztályban az adattagok mellett általában tagfüggvényeket is elhelyezünk.) Felmerül a kérdés, hogy miért volt szükséges az új kulcsszó bevezetése? A magyarázat az osztály adattagjainak (és tagfüggvényeinek) elérhetőségében keresendő.

A C nyelvvel való kompatibilitás megértésének érdekében a struktúra tagjainak korlátozás nélküli (nyilvános, *public*) elérését meg kellett tartani. Az objektum-orientált programozás alapelveinek azonban a zárt struktúra felel meg, melynek tagjait alapértelmezés szerint nem lehet elérni. Annak érdekében, hogy mindkét követelménynek megfeleljen a C++ nyelv, bevezették a `class` kulcsszót. A `class` segítségével olyan struktúrákat definiálhatunk, melynek (saját, *private*) tagjai alaphelyzetben nem érhetők el kívülről.

7.2.2 A `class` osztálytípus

```
// A könyvek véletlenszerű feltöltése
rand(unsigned(time(NULL))) ;
for (int i = 0; i < KSZAM; i++) {
    talalat[i]=false;
    sprintf(dlb[i].nev,"%03d Anonymous", i);
    sprintf(dlb[i].cim,"Nothing %03d", i);
    dlb[i].ev = 1900+rand()%106;
    dlb[i].ar = 500+rand()%6500 * 1.5;
}

// A keresett könyvek kiválogatása
int db = 0; // Nincs találat
for (int i = 0; i < KSZAM; i++)
    if (dlb[i].ev >= 1979 && dlb[i].ar <= 2005) {
        talalat[i] = true;
        ++db;
    }
// A keresés eredményének kijelzése
if (db) {
    cout<<"A talalatok száma: "<<db<<endl;
    for (int i=0; i<KSZAM; i++)
        if (talalat[i])
            cout<<dlb[i].nev<<"\t"<<dlb[i].cim<<"\t"<<
            dlb[i].ar<<endl;
    }
else
    cout<<"Nincs a feltételnek megfelelő könyv!"<<endl;
// A lefoglalt területek felszabadítása
delete [] dlb;
cin.get();
}
```

Az előző részben a **struct** típus ismertetésben elmondottak, majdnem teljes egészében megállíjják a helyüket a **class** típus esetén is. Egyetlen különbség éppen a tagok elérhetőségében rejlik. Az osztálytagok szabályozott elérése érdekében a **struct**-ra, illetve az osztály-deklarációkban **public** (nyilvános), **private** (privát) és **protected** (védezt) kulcsszavakat helyezhetünk el. Az elérhetőség megadása nélkül (alapértelmezés szerint), a **class** típusú osztály tagjai kívülről nem érhetők el (**private**), míg a **struct** típusú osztály tagjai elérhetők (**public**).

Az elmondottak fényében megállapíthatjuk, hogy az alábbi táblázat sorában megadott típusdefiniciók azonosnak tekinthetők:

```
struct comp1x {
    double re, im;
};

class comp1x {
    private:
        double re, im;
};

struct comp2x {
    public:
        double re, im;
};

class comp2x {
    double re, im;
};
```

Az osztályok használataival kapcsolatosan most csak a kezdőértékkadás kérdésével foglalkozunk. Általánosan is elmondhatjuk, hogy (**struct** vagy **class**) osztálytípusú változók definíciójában nem használunk kezdőértékeket. Meg kell jegyeznünk azonban, hogy csupa nyilvános adattaggal rendelkező osztályok esetén a kezdőértékkadás alkalmazható:

```
class comp1x {
public:
    double re, im;
};

comp1x a = {3, 4}, b;
```

A **struct** és a **class** felhasználói (absztrakt) adattípusokkal könyvünk II. részében részletesen foglalkozunk, hiszen ezeken alapulnak a C++ nyelv objektum-orientált lehetőségei.

7.2.3 A *union* típusú adatstruktúrák

A C nyelv kidolgozásakor a taktékos memóriahasználat céljából olyan lehetőségeket is beépítettek a nyelvbe, amelyek jóval kisebb jelentőséggel bírnak, mint a dinamikus memóriakezelés. Nézzük meg, miiben áll a következő két fejezetben bemutatásra kerülő megoldások lényege!

- Helyet takarítunk meg, ha ugyanazt a memóriaterületet több változó közösen használja (de nem egyidejűleg). Az ilyen változók összerendelése a C++ **struct** típusával rokon **union** (unió - egyesítés) típussal valósítható meg.

- A másik lehetőség, hogy az olyan változókat, amelyek értéke 1 bájtól kisebb területen is elfér, egyetlen bájtban helyezzünk el. Ehhez a megoldáshoz a C++ nyelv a bitmezőket biztosítja. Azt, hogy milyen (hány bites) adatok kerüljenek egymás mellé, szintén a **struct** típusal rokon bitstruktúra deklarációjával adhatjuk meg.

Az unió és a bitstruktúra megoldásokkal nem lehet jelentős memória-megtakarítást elérni, viszont annál inkább romlik a programunk hordozhatósága. A memóriáigény csökkentését célzó eljárások hordozható változata a dinamikus memóriafoglalás. Fontos megjegyeznünk, hogy az unió tagjainak nyilvános elérése nem korlátozható.

Napjainkban a **union** és bitstruktúra felhasználásának célja valamelyest megváltozott. Az unió elsősorban gyors és hatékony gépfüggő adatkonverziók megvalósítására, míg a bitstruktúrát a hardver különböző elemeinek vezérlését végző parancsszavak előállítására használjuk.

A **union** típusal igazából nincs sok dolgunk, mivel a **struct** típusal kapcsolatban ismertett formai megoldások, kezdve a deklarációtól, a pont és nyíl operátoron át egészen a struktúrákban kialakításáig, a **union** típusra is alkalmazhatók. Az egyetlen és egyben lényegi különbség az adattagok elhelyezkedése között áll fenn. Míg a struktúra adattagjai a memóriában egymás után helyezkednek el, addig az unió adattagjai közös címen kezdődnek (átlapoltak). A **struct** típus méretét az adattagok összmérete (a kiigazításokkal korrigálva) adja, míg a **union** mérete megegyezik a leghosszabb adattagjának méretével.

Az alábbi példában az uniótípus segítségével, az **unsigned long int** típusú adatokat legegyszerűbben négy **unsigned char** típusú részre bontjuk:

```
union conv {
    unsigned char ch[4];
    unsigned long szam;
};
conv u, u1={0x78, 0x56, 0x34, 0x12}; // kezdőértékadás
cout<<hex<<u1.szam<<dec<<endl;
for (int i=0; i<4; i++)
    u.ch[i]=48+i;
cout<<hex<<u.szam<<dec<<endl; // 33323130
```

A következő példánkban a **struct** és a **union** típus együttes használatát mutatjuk be. Sokszor szükség lehet arra, hogy egy állomány rekordjaiban tárolt adatok rekordonként más-más felépítésűek legyenek. Tétélezzük fel, hogy minden rekord tartalmazzon egy nevet és egy értéket, amely hol szöveg, hol pedig szám! Helytakarékos megoldáshoz jutunk, ha a struktúrán belül unióba egyesítjük a két lehetséges értéket (variáns rekord):

```

struct vrekor {
    char típus;
    char nev[25];
    union {
        char szoveg[30];
        unsigned long szam;
    }
};

vrekor vr1={'c',"ComputerBooks", "Tartsay Vilmos u. 12"};
vrekor vr2={'d',"ComputerBooks"};
vr2.ertek.szam=3751564;

cout<<vr1.ertek.szoveg<<endl;
cout<<vr2.ertek.szam<<endl;

cout<<"Név" : "<<vr1.nev<<endl;
switch (vr1.típus) {
    case 'd' :
        cout<<"Szám" : "<<vr1.ertek.szam<<endl;
        break;
    case 'c' :
        cout<<"Szöveg" : "<<vr1.ertek.szoveg<<endl;
        break;
    default :
        cout<<"Hibas adatltípus!"<<endl;
}

// nincs tagnév!
};

```

7.2.3.1 Névtelen unionok használata

A C++ nyelv lehetővé teszi, hogy a struktúrába (osztályba) névtelen uniont ágyazzunk, melynek adataitjai a struktúra (osztály) tagjaivá válnak. A fenti példát az elmondottak szerint módosítottuk:


```

switch (vr1.típus) {
    case 'd' :
        cout<<"Szam" <<vr1.szam<<endl;
        break;
    case 'c' :
        cout<<"Szoveg" <<vr1.szoveg<<endl;
        break;
    default :
        cout<<"Hibas adat!típus!"<<endl;
}

```

7.2.4 A bitmezők alkalmazása

A legtöbb programozási nyelvtől eltérően a C++ nyelv beépített módszert tartalmaz a bajton belüli bitek elérésére. Ez a megoldás több szempontból is hasznos lehet:

- Helytakarékos, bitméretű (logikai *Ki/BE*) változók használata – több változó tárolása egyetlen bájiban,
- A hardverelemek programozásához használt bit-sorozatok magas szintű kezelés.

A már megismert bitenkénti operátorok segítségével szintén elvégezhetők a szükséges műveletek, azonban a bitmezők használataával strukturáltabb kódot kapunk.

A bitmezők kezelése a bitstruktúrán keresztül valósul meg, melynek általános felépítése az alábbiakban láthatjuk:

```

struct bitstruktúra_típus {
    típus bitmezőnőv1 : bithossz;
    típus bitmezőnőv2 : bithossz;
    típus bitmezőnővN : bithossz;
};

```

A deklarációban a bitmezők neve előtt tetszőlegesen **int** típus (short, int, long, long long, valamint az **unsigned** parjait) szerepelhet. Ennek megfelelően a *bithossz* maximális értékét az adott számítógépen a legnagyobb egész típus bitmérete határozza meg. A struktúrátípusban a bitmezők és az adattagok vegyesen is szerepelhetnek.

Első példaként a *book* típusunkhoz kapcsoljunk kölcsönzési információkat!

- kölcsönözhető (1 - igen),
- kikölcsönöztek (1 - igen),
- maximum hány hétre viheto el (legfeljebb 8 hét),
- a kölcsönzés dátuma (**long**).

A fenti követelményeknek megfelelő *libbook* adatstruktúra deklarációja:

```
struct book {
    char nev [ 20 ]; // a szerző neve
    char cím [ 40 ]; // a mű címe
    int év; // a kiadás éve
    float ár; // a könyv ára
};

struct libbook {
    book könyv;
    long datum;
    unsigned kolcsonozható : 1;
    unsigned kiolcsonozható : 1;
    unsigned het : 4;
};
```

Kezdőértékek beállítása mellett definiáljunk egy változót ezzel a típussal,

```
libbook cppprog = { { "Bjarne Stroustrup",
    "The C++ Programming Language",
    2002, 23.12 },
    20041223L, 1, 0, 8 };
```

majd pedig adjunk értéket a bitmezőknek!

```
cppprog.kiolcsonozve = 1;
cppprog.kolcsonozható = 1;
cppprog.het = 3;
```

A példából kitalálunk, hogy a bitmezők segítségével egyszerűen lehet a bonyolult bitműveleteket elvégezni. A megoldás helytakarékossága szintén látható, hiszen 1 bájt használunk 3 **char** típus tárolásához szükséges 3 bájt helyett.

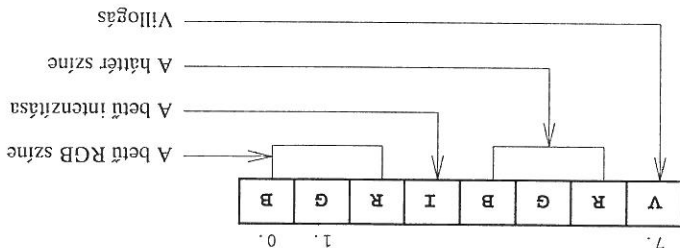
A fenti megállapítás csak akkor igaz, ha beavatkozunk a fordítóprogramok alapértelmezés szerinti memóriahatára igazításába. Például, a 32-bites fordítók „szereplik” a struktúratagokat (32-bites) duplaszórhatóra igazítani a gyorsabb elérés érdekében. Ennek következtében a fenti példában a bitenként összeállított 1 bájt számára 4 bájtot használ el a fordítóprogram működésének szabványos vezérlésére a **#pragma** előfordító direktívát használhatjuk, melynek lehetőségei teljes egészében implementációfüggők. Például, a Borland C++ Builder rendszerben a bájtíllatásra való igazításhoz az alábbi direktívákat kell megadnunk a struktúradefiníció előtt:

```
#pragma nopackwarning
#pragma pack(1)
```

A *pack* paramétereit 1,2,4,8 és 16 értékek közül választhatjuk meg. Az argumentum nélküli esetben az alapértelmezés szerinti 4 értéket használja a fordítóprogram.

Az adatterület megakartírása mellett a kódterület nagyobb lett, mintha a **char** típust használtuk volna. A futási idő szintén megnövekedett, hiszen a bitműveletek sokkal lassúbbak (nem beszélve a kiigazítás miatti lassabb memóriá-hozzáférésről), mint egy **char** típusú adattagra való hivatkozás elvégzése.

Végezetül nézzünk egy olyan alkalmazási területet (a hardver programozása), amely-nél nem vitathatók a bitmezők használatának előnyei! Tekintsük az IBM PC számító-gépek színes karakteres képernyőn való megjelenítéshez használt karakter- és attribútum-bájtot. Az attribútumbajt egy bitstruktúra, melynek felépítése az alábbi ábrán látható.



Az alábbi struktúra a karakterkódot tartalmazó és az attribútum bájtot egyaránt lefedi:

```
#pragma pack(1)
struct kepbetu {
    char kod;
    unsigned betu : 3;
    unsigned alap : 3;
    unsigned : 1; // Nem használjuk ezt a mezőt
    int : 1; // A villogást sem használjuk
};
```

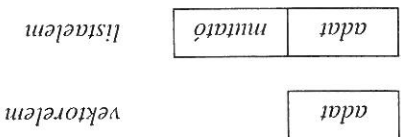
Ha a bitstruktúra deklarációjában nem adunk nevet a bitmezőnek, akkor a megadott bithosszúsági területet nem tudjuk elérni (hézagpótló bitek). Amennyiben a névtelen bitmezők hosszát 0-nak adjuk meg, akkor az ezt követő adattagot (vagy bitmezőt) **int** határra igazítja a fordítóprogram.

Végezetül foglaljuk össze a bitmezők használatának hátrányait!

- A keletkező forráskód nem hordozható, hiszen a különböző rendszerekben a bitek bájtt-, szóbeli szervezése eltérő lehet.
- A bitmezők címe nem kérdezhető le (&), hiszen nem biztos, hogy bájthátáron helyezkednek el.
- Mivel a bitmezőkkel egy lárolási egyetben több változót is elhelyezhetünk, a fordító kiegészítő kódot generál a változók kezelésére (lassul a program futása és nő a kód mérete.)

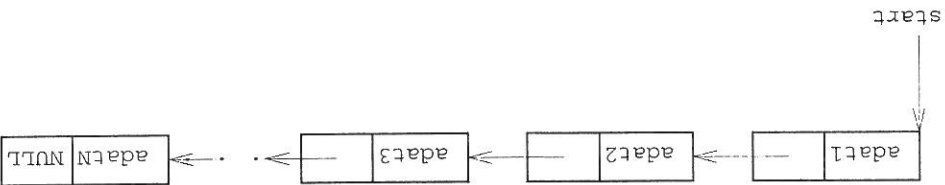
7.2.5 Esettanulmány: önhivatkozó struktúrák használata - listaszervezet

A programozási munka során kitüntetett szereppel rendelkeznek azok a megoldások, amelyekkel az adatok tárolását valamilyen listaszervezettel valósítjuk meg, a lista elemeit pedig dinamikus memóriafoglalással hozzuk létre. Nézzük meg, milyen előnyökkel jár a lista használata a vektorral (egydimenziós tömbbel) összehasonlítva. A vektor mérete definíáláskor eldől, a lista mérete azonban dinamikusan növelhető, illetve csökkenthető. Ugyancsak lényeges eltérés van az elemek beszűrése és törlése között. Míg a listában ezek a műveletek csupán néhány mutató másolását jelentik, addig a vektorban nagymennyiségű adat mozgását igénylik. További lényeges különbség van a tárolási egység, az elem felépítésében:



A vektor elemei csak a tárolandó adatokat tartalmazzák. Lista esetében azonban az adaton kívül a kapcsolati információk tárolására mutató(k)ra is szükség van.

Ezen kis bevezető után ismerkedjünk meg a legegyszerűbb listaszervezettel, a *lineáris listával*, melynek elemei egyirányú, egyszerűes láncolással (mutatókkal) kapcsolódnak egymáshoz:



A lista azonosítására a *start* mutató szolgál, ezért ennek értékét mindig meg kell őriznünk. A lista végét nulla értékű mutatóval (*NULL*) jelezzük. C++ nyelven a listaelemet a már bemutatott önhivatkozó struktúrákkal hozhatjuk létre.

Példaként tegyük önhivatkozóvá a fejezet elején deklarált *book* típust a *book * típusú kovekkezo* adattag bevezetésével!

```

struct book {
    char nev [20];
    char cim [40];
    int ev;
    float ar;
    book * kovekkezo;
};
    
```

A következőkben áttekintjük a listakezelés alapvető műveleteit.

A lista kezelése során szükségünk van segédváltozókra, illetve a lista kezdetét jelölő *start* mutatóra:

```
book *start=NULL, *elozo, *aktualis, *kovetkezo;
```

Amikor a lista adott elemével (*aktualis*) dolgozunk, szükségünk lehet a megelőző (*elozo*) és a rákövetkező (*kovetkezo*) elemek helyének ismeretére is.

A példában 10 elemet tartalmazó listát hozunk létre. A *lista felépítése* során minden egyes elem esetén három jól elkülöníthető tevékenységet kell elvégeznünk:

1. helyfoglalás (ellenőrzéssel) a listaelem számára,
2. a listaelemben tárolt adatok feltöltése,
3. a listaelem hozzáfűzése a listához (a végéhez). A hozzáfűzés során az első és a nem első elemek esetén más-más lépéseket kell végrehajtanunk.

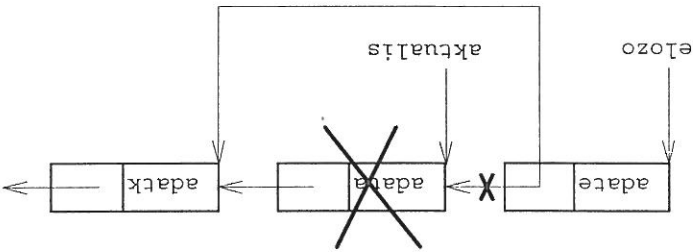
```
// A lista felépítése és az elemek véletlen feltöltése
srand(unsigned(time(NULL)));
for (int index = 0; index < 10; index++) {
    // 1. Memóriafoglalás ellenőrzéssel
    aktualis = new book;
    if (!aktualis) {
        cerr<<"\a\nMincs elég memóriai!\n";
        return -1;
    }
    // 2. A listaelem adattagjainak feltöltése
    sprintf(aktualis->nev, "%03d Anonymous", index);
    aktualis->ev = 1900 + rand() % 106;
    aktualis->ar = 500 + rand() % 6500 * 1.5;
    aktualis->kovetkezo = NULL;
    // 3. Láncolási és léptetési műveletek
    if (index == 0)
        start = aktualis;
    else {
        // az új elem láncolása az előző elemhez
        elozo->kovetkezo = aktualis;
        elozo = aktualis;
    }
}
```

A ciklus lefutása után kész van a 10 elemet tartalmazó listánk. Most következhethetnek a listakezelési műveletek. Az első ilyen művelet legyen a lista végigjárása, melynek során az egyes elemek tartalmát megjelenítjük a képernyőn. A bejárás során a *start* mutatótól indulunk, és a ciklusban mindaddig lépkedünk a következő elemre, amíg el nem érjük a lista végét jelző nulla (*NULL*) mutatót:

```
// A listában lépkedve az elemek adatainak kilírása
cout<<endl;
aktualis = start;
do
{
    cout<<aktualis->nev<<'\\t'<<aktualis->cim<<'\\t'<<
    aktualis->ev<<'\\t'<<aktualis->ar<<endl;
    // Lépés a következő elemre
    aktualis = aktualis->kovetkezo;
} while (aktualis != NULL);
```

Gyakran használt művelet a **listaelem törlése**. A példában a törendő elemet a lista-elem sorszáma alapján azonosítjuk (A sorszámozás 0-val kezdődik a *start* által kijelölt elemtől kezdődően – a programrészt nem alkalmas a 0, és az az utolsó elem törlésére!) A törlés művelete szintén három tevékenységre tagolható:

1. az adott sorszámu elem helyének megkeresése a listában,
2. a törlés elvégzése,
3. a törölt elem területének felszabadítása.



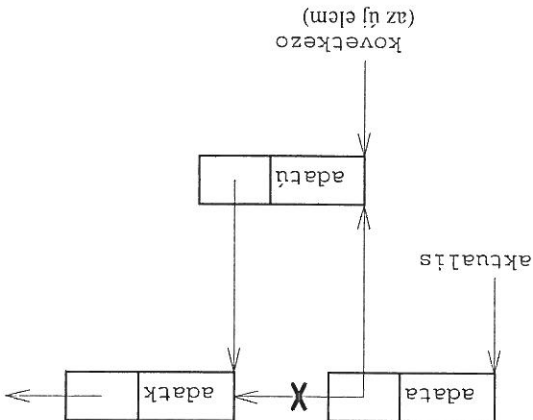
A példában a 4. sorszámu elemet töröljük a listából:

```
// A 4. sorszámu elem lokalizálása és törlése
// 1 Az elem helyének meghatározása
for (int index = 0; index<4; index++) {
    elozo = aktualis;
    aktualis = aktualis->kovetkezo;
}
// 2 A törlés - kifűzés a láncból
elozo->kovetkezo = aktualis->kovetkezo;
// 3 A terület felszabadítása
delete aktualis;
```

A 0. elem eltávolítása esetén a *start* mutató értékét a törlés előtt a *start->kovetkezo* értékre kell állítanunk. Az utolsó elem törlése esetén pedig az utolsó előtti elem *kovetkezo* tagjának nullázásáról is gondoskodnunk kell.

A törléssel ellentétes művelet az **új elem beillesztése** a listába, két meglévő elem közé. A beszűrés helyét annak az elemnek a sorszámmával azonosítjuk, amely mögé az új elemet beillesztjük. A beszűrés műveletét a következő négy lépésben végezzük:

1. a beszűrés helyének kijelölése a meglévő elem sorszáma alapján,
2. helyfoglalás az új listaelem számára,
3. a listaelem feltöltése adatokkal,
4. az elem beillesztése a sorszámmal kijelölt elem után.



A példában a 3. sorszámu elem mögé illesztünk új listaelemet:

```
// A 3. sorszámu elem lokalizálása és mögé új elem beszűrésa
// ❶ A meglévő elem helyének meghatározása
aktuális = start;
for (int index = 0; index<3; index++)
    aktuális = aktuális->következo;
// ❷ Területfoglalás az új elem számára
kovetkezo = new (nothrow) book;
if (!kovetkezo) {
    cerr<<"\a\nNincs elég memoria!\n";
    return -1;
}
// ❸ Az új elem adatainak feltöltése
strcpy(kovetkezo->nev, "!!! Anonymous");
strcpy(kovetkezo->cim, "Nothing !!!");
kovetkezo->ev = 1979;
kovetkezo->ar = 1223;
// ❹ Az új elem befűzés a láncba
kovetkezo->következo = aktuális->következo;
aktuális->következo = kovetkezo;
```

A fenti programrészlet az utolsó elem után is helyesen illeszti be az új listaelemet.

A listakezelő programok a bemutatott lépéseket használják, esetleg kibővíthetik az itt bemutatott lehetőségeket. Helyes programozási gyakorlat az, hogy kilépés előtt fel-szabadítsuk a dinamikusan foglalt memóriaterületeket. Nézzük meg, hogyan történik a *lista elemeinek megszüntetése*! Ebből a célból szintén végig kell mennünk a listán, ügyelve arra, hogy még az aktuális listaelem megszüntetése előtt kiolvassuk a követ-kező elem helyét:

```

aktuális = start;
do {
    kovetkezo = aktuális->kovetkezo;
    delete aktuális;
    aktuális = kovetkezo;
} while (kovetkezo != NULL);
start = NULL; // Nincs lista!

```


8. Függvények

A függvény a C++ program olyan névvel ellátott egysége (alprogram), amely a program más részeiből annyiszor hívható, ahányszor csak szükség van a függvényben definiált tevékenységsorozatra. A hagyományos (funkció-orientált) C++ program általában sok kisméretű, jól kézben tartható függvényből épül fel. A gyakran használt függvények lefordított kódját könyvtárakba rendezhetjük, amelyekből a szerkesztő-program a hivatkozott függvényeket beépíti a programunkba.

A függvények hatékony felhasználása érdekében a C++ nyelv lehetőséget biztosít arra, hogy a függvény bizonyos belső tárolóinak a függvényhívás során adjunk értéket. Hogy melyek ezek a tárolók és milyen típussal rendelkeznek, azt a függvény definíciójában a függvény neve után zárójelben kell megadnunk. A hívásnál pedig hasonló formában kell felsorolnunk az átadni kívánt értékeket. A szakirodalom ezekre a tárolókra és értékekre különböző nevekkel hivatkozik:

<i>a függvény-definícióban szereplő tárolók</i>	<i>a függvényhívás során megadott értékek</i>
formális paraméterek	aktuális paraméterek
formális argumentumok	aktuális argumentumok
paraméterek	argumentumok

A könyvünkben az ANSI szabvány által javasolt *paraméterek* és *argumentumok* elnevezést használjuk.

A függvényhívás során a vezérlés a hívó függvényről átkerül az aktivizált függvényhez. Az argumentumok (amennyiben vannak) szintén átadódnak a hívott függvénynek. A már bemutatott **return** utasítás végrehajtásakor, illetve a függvény fizikai végének elérésekor a hívott függvény visszatér a hívás helyére, és a **return** utasításban szereplő kifejezés, mint függvényérték (visszatérési érték) jelenik meg. A visszatérési érték nem más, mint a függvényhívás kifejezés értéke.

8.1 Függvények definíciója és deklarációja

A saját készítésű függvényeinket mindig definiálni kell. A *definíció*, amelyet csak egyszer lehet megadni, a C++ programon belül bárhol elhelyezkedhet. Amennyiben a függvény definíciója megelőzi a felhasználás (hívás) helyét akkor, ez egyben a függvény prototípusa is.

A függvény **deklarációja (prototípusa)** tartalmazza a függvény nevét, visszatérési értékének típusát, valamint információt szolgáltat a paraméterek számáról és típusáról. A prototípust a függvényhívás előtt kell elhelyeznünk a programban. A C++ fordító csak a prototípus ismeretében fordítja le a függvényhívást. (A függvény definíciója helyettesíti a prototípust.)

Az elmondottakban megtaláljuk annak magyarázatát, hogy a szabványos könyvtári függvények deklarációját tartalmazó fejelet tartalmazza a forrásfájl elején építjük be (**#include**) a programunkba. Valamely prototípus többször is szerepelhet, azonban mindegyik előfordulásnak azonosnak kell lennie.

Nézzük meg a függvény-definíció általános formáját! A függvény felsorolásban elhelyezkedő „*paraméter-deklarációs lista*” az egyes paramétereket vesszővel elválasztva tartalmazza. Minden egyes paraméter előtt szerepel annak típusa.

```
{
// a függvény törzse
<lokális definíciók és deklarációk>
    <utasítások>
}
```

A < > jelek között megadott részek hiányozhatnak a definícióból. Példaként készítsük el a nem negatív egész számokat egész kitevőre hatványozó függvényt!

```
int uihatvány( int alap, int exp )
{
    int hv = 1;
    if (exp > 0)
        for ( ; exp; exp--)
            hv*=alap;
    return hv;
}
```

A függvények definíciójában a visszatérési típus előtt megadhatjuk a tárolási osztályt is. Függvények esetén az alapértelmezés szerinti tárolási osztály az **extern**, amely azt jelöli, hogy a függvény más modulból is elérhető. Amennyiben a függvény elérhetőségét az adott modulra kívánjuk korlátozni, a **static** tárolási osztályt kell használnunk (a paraméterek deklarációjában csak a **register** tárolási osztály specifikálható). Ha a függvényt saját névtérületen szeretnénk elhelyezni, úgy a függvény definícióját, illetve a prototípusát a kiválasztott névtérület (*namespace*) blokkjába kell vinnünk. (A tárolási osztályok és a névtérületek részletes ismertetését a következő fejezet tartalmazza.)

Mint említettük a függvény prototípusa, amely általában megelőzi a függvény defini-
cióját, meghatározza a függvény nevét, visszatérési típusát (továbbá – amennyiben
megadjuk – a tárolási osztályt és a függvény attribútumait), valamint információt tar-
talmaz a paraméterekről:

```
(visszatérési típus) függvénynev (paraméter-deklarációs lista);
```

A függvény prototípusát mindig pontosvesszővel kell lezárni. Nézzük meg, milyen
mechanizmusok érvenyesülnek a program fordításakor a prototípust használata során!

- A prototípus definiálja a függvény visszatérési típusát, amennyiben az eltér az **int** típustól.
- Az argumentumok konverziója a prototípusban definiált típusoknak megfele-
lően, nem pedig az automatikus konverziók szerint megy végbe.
- A paraméterlista és az argumentumlista összevetésével a fordító ellenőrzi a
paraméterek számának és típusainak összeférhetőségét.
- A prototípus függvénymutató inicializálására is felhasználható.

A prototípus gyakorlatilag megegyezik a függvénydefiníció első sorával (a függvény-
fejtel), amelyet pontosvesszővel zárunk. A prototípus általában csak a paraméterek tí-
pusát tartalmazza, amennyiben a paraméterek nevét is megadjuk, akkor azokat figyel-
men kívül hagyja a fordító:

```
(visszatérési típus) függvénynev ((típuslista));
```

Az elmondottak alapján az alábbi két prototípus megegyezik:

```
int uihatvány ( int, int );
```

```
int uihatvány( int alap, int exp );
```

Felhívjuk a figyelmet arra, hogy a paraméterrel nem rendelkező függvények prototípu-
sat eltérő módon értelmezi a C és a C++ nyelv:

Deklaráció	C értelmezés	C++ értelmezés
<code>típus f();</code>	<code>típus f(...);</code>	<code>típus f(void);</code>
<code>típus f(...);</code>	<code>típus f(...);</code>	<code>típus f(...);</code>
<code>típus f(void);</code>	<code>típus f(void);</code>	<code>típus f(void);</code>

A C++ nyelv lehetővé teszi, hogy a legalább egy paramétert tartalmazó paraméterlistát
a ... deklaráció zárja. Az így definiált függvény legalább egy, de különben tetszőleges
számú és típusú argumentummal hívható. Példaként tekintsük a *printf()* függvény
prototípusát!

```
int printf( const char * formatum, ... );
```

8.2 A függvények paraméterezése és a függvényérték

A C++ függvény-definícióban szereplő paraméterlistában minden paraméter előtt ott áll a paraméter típusa. A paraméterek deklarációs sorrendje követi a paraméterek sorrendjét, és semmilyen összevonás sem lehetséges.

```
int uihativany( int alap, int exp ) {
    // ...
}
```

A deklarált paramétereket a függvényen belül a függvény lokális változóiként használhatjuk, azonban a függvényen kívülről nem érhetők el. A paraméterek típusa a skálár (**bool**, **char**, **wchar_t**, **short**, **int**, **long**, **long long**, **float**, **double**, felsorolási, referencia és mutató), valamint a struktúra-, unió-, osztály- és tömbtípusok közül kerülhet ki.

A visszatérési típus meghatározza a függvényérték típusát, amely tetszőleges skálár vagy struktúrált (**struct**, **class**, **union**) típus lehet. (Nem lehet azonban tömbtípus.)

A függvény a **return** utasítás feldolgozásokor ad vissza értéket, amelyet (ha szükség-ges) a visszatérési típusra konvertál. A visszaadott érték az utasításban szereplő kifejezés értéke:

```
return kifejezés;
```

Ha a függvény definíciójában nem adjuk meg a visszatérési típust, akkor alapértelmezés szerint **int** típusú lesz a függvényérték.

A függvényen belül tetszőleges számú **return** utasítás elhelyezhető. Az alábbi faktoriális számító függvényekben egy, illetve két **return** utasítást használunk. (A függvények önmagukat hívó, rekurzív függvények.)

```
int fact1( int n )
{
    return (n>1) ? n*fact1(n-1) : 1;
}

int fact2( int n )
{
    if (n>1)
        return n * fact2(n-1);
    else
        return 1;
}
```

A **void** típus felhasználásával olyan függvényeket is készíthetünk, amelyek nem adnak vissza értéket. (Más programozási nyelveken ezeket az alprogramokat eljárásoknak nevezzük.) Ebben az esetben a függvényből való visszatérésre a **return** utasítás

kifejezés nélküli alakját használjuk. A **void** függvényekben gyakran a függvény törzset záró kapcsolós zárójellet használjuk visszatérésre. Az alábbi *sorinita* függvény a megadott karaktert adott számszor kiírja egymás mellé:

```
void sorinita(int db, char ch)
{
    for (register int i=0; i<db; i++)
        cout<<ch;
}
```

A különböző C++ változatokban a visszatérési típus után különböző függvényattribútumok is szerepelhetnek, amelyek a függvény alapértelmezés szerinti működését módosítják. A legtöbb Windows alatt működő C++ fejlesztőrendszerben az alábbi attribútumokat (módosítókat) használhatjuk:

- **pascal** - Pascal függvényhívási konvenciók,
- **fortran** - Fortran függvényhívási konvenciók,
- **cdecl** - C függvényhívási konvenciók,
- **fastcall** - Delphi függvényhívási konvenciók,
- **stdcall** - Windows által alkalmazott függvényhívási konvenciók.

8.3 A függvényhívás

A függvényhívás olyan kifejezés, amely átadja a vezérlést és az argumentumokat (ha vannak) az aktivizált függvénynek. A függvényhívás általános alakja:

kifejezés1((*kifejezés2*))

ahol a *kifejezés1* a függvény neve (vagy a függvény címét szolgálító kifejezés), míg az opcionálisan megadható *kifejezés2* az argumentum-kifejezések vesszővel tagolt listája.

```
int fV ( int a, float b ) : // prototípus
x = fV ( 1, 2.23 ) ; // függvényhívás
```

Az argumentumok kiértékelésének sorrendjét nem definiálja a C++ nyelv. Egyetlen dolgot garantál mindössze a függvényhívás operátora, hogy mire a vezérlés átködik a hívott függvénynek, az argumentumlista teljes kiértékelése (a mellélékhataásokkal együtt) végbemegy. Azon függvények esetén, ahol a prototípusban a paraméterlista helyén **void** kulcsszó szerepel, a híváskor nem adható meg egyetlen argumentum sem:

```
int fV(void) ;
// prototípus
x = fV() ; // függvényhívás
```

Mint láttuk a függvényhívásnál használt argumentumok skalar, struktúra, osztály vagy unió típusúak lehetnek. A C++ nyelvben az argumentumok érték szerint adódnak át a hívott függvényeknek. Ez azt jelenti, hogy az argumentum másolatát veszi fel a megfellelő paraméter értékékként. Ennek következtében, ha függvényben belül a paraméteren valamilyen műveletet végzünk, annak nincs kihatása a híváskor megadott argumentumra.

Ezek után felmerül a kérdés, hogyan lehet C++-ban olyan függvényt készíteni, amely felcseréli két egész típusú változó értékét? Az érték szerint argumentum-átadás látszólag ezt nem teszi lehetővé. Ha azonban az átadott érték valamely változónak a címe vagy referenciája, akkor ezek felhasználásával lehetséges nyílik arra, hogy a függvényből „kihívatkozva” megváltoztassuk a változó értékét. Nézzük példaként az egész változók értékének felcserélését megvalósító programot!

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

void cserel(int *, int *); // prototípusok

void cserel2(int &, int &);

void main()
{
    int x=7, y=30;
    cserel(&x, &y); // függvényhívások
    cserel2(x, y); // x=7, y=30
}

// A cserel függvény definíciója
void cserel ( int * p, int * q )
{
    int sv = *p;
    *p = *q;
    *q = sv;
}

// A cserel2 függvény definíciója
void cserel2 ( int & a, int & b )
{
    int sv = a;
    a = b;
    b = sv;
}
```

A *cserel()* függvény argumentumai nem a változók értékei, hanem a változók címei (&x és &y), és ezek adódnak át, amiket egészen mutató pointerekbe, mint paraméterekbe, vesszünk át (*int *p és int *q*). A cserét ezek után a **p* és a **q* tárolók közötti végzetjük el, egy *sv* segédváltozó bevezetésével.

A referencia típus igazi lehetőséget a hivatkozás szerinti paraméterátadás jelenti. A `cserve2()` függvény referencia típusú paraméterei (`int &a`, `int &b`) a függvényen belül a híváskor átadott argumentumokra (x, y) hivatkoznak. A függvény blokkjában az a paraméter az x változó, a b paraméter pedig az y változó második nevéként jelenik meg. (A háttérben a fordító valójában címeket másol, azonban a forrásprogramból ez nem látható.)

Nézzünk néhány érdekes példát a mutatók és a referenciaik paraméterezésében való használatára! Az elsöben egy **void** típusú függvény segítségével megvalósítjuk az egész típusú operandussal rendelkező „címe” operátort (&). A megoldásban az okoz nehézséget, hogy mutatóra mutató pontert (**) kell ahhoz átadnunk a `cime` függvénynek, hogy az a függvényen belül értéket kapjon. Másrészt az egész változó címének átadásáról (*) is gondoskodnunk kell.

```
void cime(int ** pp, int * p) {
    *pp = p;
}
```

```
void main()
{
    int a = 79120;
    int *ap;
    cime(&ap, &a); // Mint az ap = &a
    *ap += 23; // Az a értéke 79143 lesz!
}
```

A feladat jobb megoldásához jutunk, ha a mutatói (`int *`) és az egész változót egyaránt referenciaival adjuk át a függvénynek:

```
void cimer(int * &pp, int &p) {
    pp = &p;
}
```

```
void main()
{
    int a = 79120;
    int *ap;
    cimer(ap, a); // Mint az ap = &a
    *ap += 23; // Az a értéke 79143!
}
```

Felhívjuk a figyelmet arra, hogy referenciahoz sem mutatói (`int &*`), sem referenciai (`int &&`) nem készíthetünk.

A következőben olyan függvényt mutatunk be, amely mutató típusú visszatérési értékkel rendelkezik. A függvény az argumentumként átadott címet egyszerűen visszaadja függvényértékként:

```
int * kozvetlt1( int * p)
{
    return p;
}

void main()
{
    int a=79120;
    *kozvetlt1(&a) += 23; // mint az a += 23;
    // az a értéke 79143 lesz!
}
```

Az előző függvény helyett sokkal biztonságosabb az alábbi használat, amely referenciát fogad a paraméterében, és ezt függvényértékként vissza is adja:

```
int & kozvetlt2( int & r)
{
    return r;
}

void main()
{
    int a=79120;
    kozvetlt2(a) += 23; // mint az a += 23;
    // az a értéke 79143 lesz!
}
```

Figyeljünk arra, hogy függvényen belül definiált lokális tárolók (*auto*) címének, illetve referenciájának kiadása a függvényből súlyos következményekkel járhat! Ennek oka, hogy a függvényből kilépve ezek a változók megszűnnek létezni, így címükkel (referenciájukkal) érvénytelen memóriaterületre hivatkozunk.

8.4 Különböző típusú paraméterek használata

A függvények készítése során a legkülönbözőbb típusú adatok átadására lehet szükség. Lényeges, hogy mindig az igényeknek megfelelően válasszuk meg a függvényeink paramétereit. Ebben segítségünkre lehetnek az alábbiak, ahol áttekinjük a különböző típusú paraméterek használatának szabályait.

8.4.1 Aritmetikai típusú paraméterek

Az alábbiakban elmondottak a **bool**, **char**, **wchar_t**, **int**, **enum**, **float** és **double** típusokra, illetve ezek típusmódosítókkal (**signed**, **unsigned**, **short**, **long**, **long long**) ellátott változataikra egyaránt érvényesek. A fenti típusokkal minden további nélkül deklarálhatunk paramétereket, illetve függvények típusaként is megadhatjuk őket.

Egyszerűség kedvéért tekintsük a két **double** szám összegzését végző függvény külön-böző változatait! Mivel a függvényünk egyetlen értéket szolgáltat eredményül, ezt megteheti függvényértékként (*osszeg1()*), illetve egy referencia (vagy pointer) paramé-teren keresztül (*osszeg2()*).

```
double osszeg1(double a, double b)
{
    return a+b;
}

void osszeg2(double a, double b, double &c)
{
    c=a+b;
}
```

Természetesen eltérő módon kell hívunk a két függvényt:

```
cout<<osszeg1(10,20.3)<<endl;
double d;
osszeg2(10,20.3,d);
cout<<d<<endl;
```

8.4.2 Felhasználoi típusú paraméterek

A C++ nyelv felhasználoi típusainak (**struct**, **class**, **union**) paraméterlistában, illetve függvényértékként való felhasználására az aritmetikai típusoknál ismertett szabályok érvényesek. Ennek alapja, hogy a nyelv definiálja az azonos típusú osztályok, illetve uniók közötti értékadást. Az ok, amiért mégis részletesen foglalkozunk ezzel a kérdés-sel, az objektum-orientált programépítés eszköztárában keresendő.

A felhasználoi típusú argumentumok függvénynek való átadása során az érték szerinti, a referenciával, illetve a mutató segítségével megvalósított megoldások között választ-hatunk. Az szabványos C++ nyelvben a függvény visszatérési értéke felhasználoi típu-sú is lehet. A lehetségesek közül általában ki kell választanunk az adott feladathoz leg-jobbban illeszkedő megoldást. A választásnál figyelembe vehetjük, hogy a memória-igény, illetve a futási idő szempontjából melyik megoldás a leghatékonyabb.

Példaként vegyük a komplex számok tárolására alkalmas struktúrát!

```
struct complex {
    double re, im;
};
```

Készítsünk függvényt két komplex szám összeadására (*csun1()*), amelyben a tagok és az eredmény tárolására szolgáló struktúrát mutatójuk segítségével adjuk át a függ-vénynek! Mivel a bemenő paramétereket nem kívánjuk a függvényen belül megváltoz-tatni, **const** típuselőírást használunk.

```

void csuml(const complex *pa, const complex *pb, complex *pc)
{
    pc->re = pa->re + pb->re;
    pc->im = pa->im + pb->im;
}

```

A második függvény (*csum2()*) visszatérési értéként szolgáltatja a két érték szerint adott komplex szám összegét. Az összegezést egy lokális struktúrában végezzük el, melynek értékét a **return** utasítással adjuk vissza.

```

complex csum2(complex a, complex b)
{
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

```

A második megoldás természetesen sokkal biztonságosabb, és sokkal jobban kifejezi a művelet lényegét, mint az első. Minden más szempontból (memóriaigény, sebesség) az első függvényt kell választanunk. A referenciátípus használatával azonban olyan megoldáshoz juthatunk, amely magában hordozza a *csum2()* függvény előnyös tulajdonságait, azonban az *csuml()* függvényel is felveszi a versenyt.

```

complex csum3(const complex &a, const complex &b)
{
    complex c;
    c.re = a.re + b.re;
    c.im = a.im + b.im;
    return c;
}

```

A konstans referenciák az egyirányú paraméterátadásra utalnak, melynek során az argumentum-struktúra tartalma nem másolódik, csak hivatkozunk rá.

A három különböző megoldáshoz két különböző hívási mód tartozik. Az alábbi programrészt mind a három összegző függvény hívását tartalmazza:

```

void main()
{
    complex c1 = {4, 7}, c2 = {26, 13}, c3;

    csuml(&c1, &c2, &c3); // mindhárom argumentum pointer
    c3 = csum2(c1, c2); // két struktúra az argumentum
    c3 = csum3(c1, c2); // két struktúra-referencia az argumentum
}

```

8.4.3 Tömbök átadása függvénynek

A következőkben megnézzük, hogy milyen lehetőségeket biztosít a C++ nyelv tömbök függvénynek való átadására. Már a legegyszerűbb le kell szögeznünk, hogy tömböt *nem lehet érték szerint* (a teljes tömb átmásolásával) függvénynek átadni, illetve függvényértéként megkapni. Sőt különbség van az egydimenziós (vektorok) és a többdimenziós tömbök argumentumként való átadása között.

8.4.3.1 Vektorargumentumok

Egydimenziós tömbök (vektorok) függvényargumentumként való megadása esetén a tömb első elemére mutató pointer adódik át. Más szavakkal a `T[]` típusú vektorargumentum `T*` típusú mutatóként adódik át a hívott függvénynek. Ebből a megoldásból viszont az is következik, hogy a vektor elemein, a függvényen belül végrehajtott változtatások a függvényből való visszatérés után is érvényben maradnak.

A vektorok átadásának fenti módszere elegendő ahhoz, hogy a vektor elemeit elérjük (gondoljunk a vektor és a mutatók közötti analógiára), azonban semmilyen információ nem jut el a függvényhez a vektor méretével (eleminek számával) kapcsolatban. Ezt az adatot egy második paraméter felhasználásával adhatjuk meg. Kivételt képeznek azok az egydimenziós karaktertömbök, amelyekben stringet tárolunk, hisz karakter-sorozatok esetén a memória tartalmazza a string végét jelölő 0-as bájtot.

Nézzünk néhány jellegzetes megoldást a vektorokat feldolgozó függvények kialakítására! Az alábbi példák mindegyike az adott egész típusú vektor elemeinek átlagát határozza meg, és adja vissza függvényértékként. A hívás bemutatásához az alábbi 10-elemű vektort használjuk:

```
int a[10] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
```

Az első példában `int*` mutatóként fogadjuk a vektor kezdőcímét, és a függvény törzse-ben is pointerműveletekkel érjük el az elemeket. Amennyiben a vektor elemeit nem kívánjuk megváltoztatni, konstansra mutató pointert definiálunk paraméterként: (`const int * vektor`). Ha emellett a mutató állandóságot is biztosítani szeretnénk, konstansra mutató konstans pointerben fogadjuk a tömb címét: (`const int * const vektor`).

```
double atlag1 (const int * vektor, int n)
{
    long sum = 0;
    for (int i = 0; i < n; i++)
        sum += *(vektor+i);
    // vagy: sum += vektor[i];
    return (double)sum / n;
}
```

A következő példában `int[]` típusú deklaráljuk a függvény paraméterét. Az `int[]` deklaráció csak annyit közöl a vektortól, hogy egész elemeket tartalmaz, így hatasta megegyezik az `int * const` deklarációéval. (Ha a zárójelék között valamilyen egész kifejezést adunk meg, akkor azt figyelmen kívül hagyja a fordító.) Az elemek állandóságát itt is a `const` típusmódosítóval biztosíthatjuk: `const int vektor[]`.

```
double atlag2 (const int vektor[], int n)
{
    long sum = 0;
    for (int i = 0; i < n; i++)
        sum += vektor[i];
    return (double)sum / n;
}
```

Mindkét függvényt ugyanúgy kell hívni:

```
double b1 = atlag1 (a, 10);
double b2 = atlag2 (a, 10);
```

A vektor címet és méretét más módon is megadhatjuk, például egy struktúrában egye-síve a két adatot:

```
struct vec {
    int *ptr; // a vektor címe
    int size; // az elemek száma
};
```

Ezt a struktúrát értéként adjuk át az átlagot számító függvénynek, melynek proto-típusa:

```
double atlag3 (vec vs);
```

A hívás módja természetesen eltér az előző függvényekétől, hisz a hívás előtt fel kell töltenünk egy `vec` típusú struktúrát a szükséges adatokkal:

```
vec v = {a, 10};
double b2 = atlag3 (v);
```

A programozás során gyakran mutatókat tartalmazó vektort használunk kétdimenziós tömbök helyett. Az ilyen vektor argumentumként való átadása az előzőekben bemutatott megoldások bármelyikével elvégezhető. Az alábbi függvények az átadott string-tömb (karakterre mutató pointer) elemeit egymás mellé írja, szökőzssel tagolva. (A stringtömbben a stringeket `NULL` pointer zárja.) A megoldást pointeres `printf()` és vektoros `printf2()` felfogásban az alábbi program tartalmazza.

```

#include <iostream>
using namespace std;

void print1(char **p)
{
    while (*p)
        cout<<*p++<<' ';
    cout<<endl;
}

void print2(char *p[])
{
    int i=0;
    while (p[i])
        cout<<p[i++]<<' ';
    cout<<endl;
}

void main()
{
    char *Desc[] = {"Cogito", "ergo", "sum.", "NULL"};
    print1(Desc);
    print2(Desc);
    cin.get();
}

```

8.4.3.2 Kétdimenziós tömbargumentumok

A kétdimenziós tömb elnevezés alatt most csak a fordító által (statikusan) létrehozott tömböket értjük:

```
int a[3][4];
```

A tömb elemére való hivatkozás $(a[i][j])$ mindig átirható a $((int*)a+(i*4)+j)$ formula szerint (ezt teszik a fordítók is). Ebből a kifejezésből is látszik, hogy a kétdimenziós tömb második dimenziója (4) alapvető fontossággal bír a fordító számára, míg a sorok száma tetszőleges lehet.

Végso célunk olyan függvény készítése, amely tetszőleges méretű kétdimenziós tömb elemeit mátrixos formában jeleníti meg. Elso lépésként írjuk meg a függvény 3x4-es tömbök megjelenítésére alkalmas változatát!

```

void printMat34(const int matrix[3][4]) // hívás: printMat34(a);
{
    for (int i=0; i<3; i++)
        for (int j=0; j<4; j++)
            cout<<'\\t'<<matrix[i][j];
    cout<<endl;
}

```

A kétdimenziós tömb is a terület kezdőcímét kijelölő mutatóként adódik át a függvénynek. Azonban a fordító az elemek elérése során

```
* (int *)mx + (i*4)+j )
```

figyelembe veszi azt, hogy a sorok 4 elemet tartalmaznak. Ezért a fenti függvény egy-szerűen átalakítható olyan függvényre, amely *mx4*-es tömb kitérására alkalmas, csak a sorok számát kell átadni második argumentumként:

```
void PrintMatn4 (const int matrix[][4], int n)
// hívás: PrintMatn4(a, 3);
{
    for (int i=0; i<n; i++)
        for (int j=0; j<4; j++)
            cout<<'\\t'<<matrix[i][j];
    cout<<endl;
}
```

Arra azonban nincs lehetőség a C++ nyelvben, hogy a második dimenziót is elhagyjuk, hiszen akkor a fordító nem képes a tömb sorait azonosítani. Egyetlen dolgot tehetünk az általános megoldás megvalósításának érdekében, hogy átvesszük a tömbterület elérését a fordítótól (a fenti kifejezés felhasználásával):

```
void PrintMatnm(void *mx, int n, int m)
// hívás: PrintMatnm(a, 3, 4);
{
    for (int i=0; i<n; i++)
        for (int j=0; j<m; j++)
            cout<<'\\t'<<*(int *)mx+i*m+j);
    cout<<endl;
}
```

8.4.4 Sztringargumentumok

A karaktersorozat-argumentumok a már ismertett módon, egydimenziós karaktertömbökként adódnak át a függvényeknek. Az ok, amiért mégis külön kifejezést szentelünk e témának, a sztringek feldolgozásánál használt fogások bemutatása. A sztringek kezelése során alapvetően két megközelítési módot használhatunk. Az első esetben mint vektort kezeljük a sztringet (indexek alapján), a másik megközelítés szerint mutató segítségével végezzük el a szükséges műveleteket.

Az első példában mindkét megoldást bemutatjuk az *strcpy()* (sztringmásolás) könyvtári függvény működését megvalósító függvényben. A további példákban azonban fel-

A szabványos könyvtár *strcpy()* függvénye

```
char * strcpy (char * celsr, const char * forrsstr);
```

a *forrsstr* sztring tartalmát átmásolja a *celsr* sztringbe, és a célstringre mutató pointerrel tér vissza. A műveletet mutatókkal megvalósító függvény:

```
char * strcpy(char *p, const char *q)
{
    char *s = p; // a célterület
    while (*p++ = *q++);
    return s;
}
```

A *strcpy()* függvényben a *p* és *q* mutatókkal dolgozunk. Ahhoz, hogy a célterületet kijelölő mutatót a függvény végén vissza tudjuk adni, az *s* segédváltozóban megőrizzuk azt. A függvénynek nincs külön paramétere, amely a forrássztring hosszát tartalmazna. Erre nincs is szükségünk, hisz a sztringet záró '\0' karakter (0-as bájt) vizsgálatával végig tudunk lépkedni a sztringen. A sztringeken való végighaladást (++) és a karakterek másolását (=) egyetlen **while** ciklusba sűrítettük. A ciklus akkor áll le, amikor a 0-as bájt is átmásolódott. A ciklust természetesen kevésbé tömör formában is fel lehet írni:

```
while (*p = *q) {
    p++;
    q++;
}
*p = 0;
```

Armenyiben vektorként kívánjuk a sztringet feldolgozni, szükségünk van egy indexváltozóra, amellyel a vektorokat indexeljük:

```
char * strcpy(char p[], const char q[])
{
    int i=0; // indexeléshez
    while (p[i] = q[i]) i++;
    return p;
}
```

A másolást leállító feltétel szintén a sztringet záró 0-as bájt elérése. Ehhez a bájtához az indexváltozó (!) léptetésével jutunk el. Összehasonlítás kedvéért írjuk fel a másoló ciklus kevésbé tömör alakját is!

```
for (int i=0; q[i]; i++)
    p[i] = q[i];
p[i] = 0;
```

Látniuk, hogy mutatók segítségével tömörebben lehet megfogalmazni a feladat megoldását. Azonban ez a tömörség a program olvashatóságát lényegesen romlítja.

A következő függvény az első argumentumként átadott sztringhez hozzáadja a második argumentumban megadott sztring tartalmát:

```
char * strcat(char *p, const char *q)
{
    char *s = p;
    while (*p) p++; // lépkedés a célsztring végére
    while (*p++ == *q++); // másolás
    return s;
}
```

A `index()` függvény az `s1` sztringben megkeresi az `s2` sztring első előfordulását, és visszatér a helyet azonosító indexszel. A `-1` függvényérték azt jelzi, hogy nem található meg az `s2` az `s1`-ben:

```
int index(const char s1[], const char s2[])
{
    int j, k;
    for (int i=0; s1[i]; i++) // Lépkedés az s1-ben
    {
        // Az s1 i. pozíciójától van-e az s2
        for (j=1, k=0; s2[k] && s1[j] == s2[k]; j++, k++);
        // Ha a leállítás feltétele az s2 vége - benne van!
        if (s2[k] == '\0') return i;
    }
    return -1; // Nincs benne
}
```

8.4.5 A függvény, mint argumentum

Matematikai alkalmazások készítése során jogos igény, hogy egy jól megvalósított algoritmust különböző függvények esetén tudjuk használni. Ehhez a szükséges függvényt mint argumentumot kell átadni az algoritmust megvalósító függvénynek.

8.4.5.1 A függvény típus és a `typedef`

Mielőtt megismerkednénk a függvényre mutató pointerekkel, nézzük meg a `typedef` tárolási osztály felhasználását függvények esetén! A `typedef` segítségével a függvény típusát egyetlen szinonim névvel jelölhetjük. A függvény típus deklarálja azt a függvényt, amely az adott számú és típusú paraméterhalmazzal rendelkezik, és a megadott adattípussal tér vissza. Tekintsük például a faktoriális számító függvényt, melynek prototípusa és definíciója:


```
unsigned long fakt(int); // prototípus
{
    unsigned long f = 1;
    for (; n > 0; n--) f *= n;
    return f;
}
```

Most pedig vegyük a függvénydefiníció fejsorát, tegyük elé a **typedef** kulcsszót, majd pontosvesszővel zárjuk le azt! A keletkező új típus neve legyen *faktv*!

```
typedef unsigned long faktv(int n);
```

A *faktv* típus felhasználásával a *fakt()* függvény prototípusa az alábbi alakban írható fel:

```
faktv fakt; // prototípus
```

8.4.5.2 Függvényre mutató pointerek

A C++ nyelvben a függvényneveket kétféle módon használhatjuk. A függvénynevet a függvényhívás operátor bal oldali operandusaként megadva függvényhívás kifejezést kapunk

```
fakt(7)
```

melynek értéke a függvény által visszaadott érték. Ha azonban a függvénynevet önállóan használjuk

```
fakt
```

akkor egy mutatóhoz jutunk, melynek értéke az a memóriacím, ahol a függvény kódja elhelyezkedik (kódpointer), típusa pedig a függvény típusa.

Definiáljunk egy olyan mutatót, amellyel a *fakt()* függvényre mutathatunk, vagyis értékként felveheti a *fakt()* függvény címet! A definíciót egyszerűen megkapjuk, ha a *fakt* függvény fejsorában szereplő nevet a *(*fptr)* kifejezésre cseréljük:

```
unsigned long (*fptr)(int);
```

Az *fptr* olyan pointer, amely **unsigned long** visszatérési értékkel és egy **int** típusú paraméterrel rendelkező függvényre mutathat.

A definíció azonban sokkal olvashatóbb formában is megadható, ha használjuk a **typedef** segítségével előállított *faktv* típust:

```
faktv *fptr;
```

Az *fptr* nemcsak mutatathat, hanem rá is mutat az alábbi értékadás végrehajtása után a

fakt() függvényre:

```
fptr = fakt;
```

Ezek után a *fakt* függvény az *fptr* mutató felhasználásával indirekt módon is hívható:

```
f10 = (*fptr) (10);      vagy      f10 = fptr (10);
```

A **fptr* kifejezést azért kell zárójelben használnunk, mert a függvényhívás operátora erősebb precedenciájú az indirekt hívalkozás operátoránál. Az *fptr* mutató természetesen tetszőleges, a *fakt()* függvénnnyel megegyező típusú függvény címet felveheti.

Hasonló módon kell eljárunk, ha a függvényhez referenciát akarunk készíteni. Ebben az esetben a kezdőértéket már a definíció során meg kell adni:

```
unsigned long (& fref) (int) = fakt;
```

vagy

```
faktf & fref = fakt;
```

A függvényhívás a referencia felhasználásával:

```
f10 = fref(10);
```

A függvény neve és a függvényre mutató pointer között hasonló összeállítás van, mint a tömb neve és a tömbelem típusára mutató pointer között. (Mind tömbnév, mind pedig a függvénynév konstans mutatóként viselkedik.) Csak emlékeztetőül a tömb és a mutatók közötti kapcsolat:

```
int a[10], *p = a;
```

```
a[0] = a[1];      *(a+0) = *(a+1);
p[0] = p[1];      *(p+0) = *(p+1);
```

Nézzük meg, mi a helyzet az *fptr* függvény és a rá hivatkozó *pfp* esetén:

```
void fv (int);
void (*pfv) (int) = fv;
```

```
// A lehetséges függvényhívások:
fv(2);
pfv(2);
(*pfv) (2);
```

Az elmondottak alapján megérthetjük a *gsort()* könyvtári függvény prototípusát:

```
void gsort(void *base, size_t nelem, size_t width,
int (*fcmp) (const void *, const void *));
```

A függvényel *base* címen kezdődő, *nelem* elemszámaú, elemenként *widih* bajtot foglaltó tömböt rendezhetünk sorba. A rendezés során hívott összehasonlító függvényt megadunk, azaz *fcn* paraméterben.

Az alábbi példában a *qsort()* függvényt egy egész, illetve egy sztringtömb rendezésére használjuk:

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

int icmp(const void *p, const void *q) {
    return *(int *)p - *(int *)q;
}

int strcmp(const void *p, const void *q) {
    return strcmp((char *)p, (char *)q);
}

void main()
{
    int m[5] = {3, 0, 6, 1, 7};
    char nevek[3][20] = {"Dennis Ritchie", "Bjarne Stroustrup",
        "Ken Thompson"};

    qsort(m, 5, sizeof(int), icmp);
    for (int i = 0; i < 5; i++)
        cout << m[i] << endl;

    qsort(nevek, 3, 20, strcmp);
    for (int i = 0; i < 3; i++)
        cout << nevek[i] << endl;
    cin.get();
}
```

A következő példaprogramokban szereplő *tabloz()* függvény teljesen *double* típusú paraméterrel és *double* visszatérési értékkel rendelkező függvény értékeinek táblázatos megjelenítésére alkalmas. A *tabloz()* függvény paraméterei között szerepel még az intervallum két határa és a lépésköz.

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

// A tabloz() függvény prototípusa
void tabloz(double (*)(double), double, double, double, double);

double sqr(double); // a saját függvény prototípusa
double sqrt(double); // a könyvtári függvény prototípusa
```

```

int main() {
    cout<<"\n\ naz sqrt() függvény értékei ([-2,2] dx=0.5)"<<endl;
    tabloz(sqrt, -2, 2, 0.5);
    cin.get();

    cout<<"\n\ naz sqrt() függvény értékei ([0,2] dx=0.2)"<<endl;
    tabloz(sqrt, 0, 2, 0.2);
    cin.get();

    // A tabloz() függvény definíciója
    void tabloz(double (*fp)(double), double a, double b,
        double x;
        for (x=a; x<=b; x+=lepes)
            printf("%13.5f \t %15.10f\n", x, (*fp)(x));
    }

    // Az sqrt() függvény definíciója
    double sqrt(double x) {
        return x * x;
    }

```

8.4.5.3 Függvényre mutató pointerek tömbje

C++ nyelven gyakran használ, hasznos adatstruktúra a függvényre mutató pointerek tömbje, amely szintén lehet egy- vagy többdimenziós. Előszörban olyan esetekben használjuk ezeket a tömböket, amikor valamilyen választás (például menüből menü-pont kiválasztása) eredményét gyorsan kívánjuk feldolgozni.

Tegyük fel, hogy a programunk vezérlését egyszerű menüvel szeretnénk megoldani!

```

1. Beolvas
2. Szamol
3. Kijir
-----
0. Kilep

```

A hagyományos megoldás során ciklusban fogadjuk a karaktereket, és az **if** vagy a **switch** utasítással kiválasztjuk a megfelelő esetet

```

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

```

```

// A menü definíciója
char * menu[] = { "\n",
    "1. Beolvas",
    "2. Szamol",
    "3. Kijir",
    "-----",
    "0. Kilep",
    "\n",
    NULL
};

```

```
// A függvények prototípusa
void beolvas(void);
void szamol(void);
void kilir(void);
// globális változók, közös használatra
int a, b, c;
```

```
int main()
{
    char ch, **p;
    do {
        // a menü kilírása
        p = menu;
        while (*p)
            cout<<*p++<<endl;
        // A választás feldolgozása
        cin>>ch; cin.get();
        switch (ch) {
            case '0': break;
            case '1': beolvas(); break;
            case '2': szamol(); break;
            case '3': kilir(); break;
            default: cout<<'\a';
        }
    } while (ch != '0');
```

```
// A menüpontnak megfelelő függvények definíciója
void beolvas(void) { // biztonságos adatbevitel
    char sor[64], ch;
    cout<<"Kérek két számot [a,b]: ";
    cin.getline(sor, 64);
    istringstream ist(sor);
    ist>>a>>ch>>b;
}

void szamol(void) {
    c = a + b;
}

void kilir(void) {
    cout<<a<<" + "<<b<<" = "<<c<<endl;
}
```

Nézzük meg, hogyan működösül a *main()* függvény a mutatóimban bevezetésével! A fenti három (azonos típusú) függvény címének tárolására használható *menupont* vektor definíciója (kezdőértékek megadásával):

```
void (*menupont[3]) (void) = { beolvas, szamol, kilir };
```

A *main()* függvény sokkal rövidebb és áttekinthetőbb lett. Meg kell jegyeznünk, hogy minél több a menüpont (például kétdimenziós menü esetén), annál hatékonyabb ez a megoldás. A tömbelem által kijelölt függvény (például a *szamol*) hívása:

```
(* menupont[1])(); vagy menupont[1]();
```

A módosított *main()* függvény:

```
void main()
{
    char ch, **p;
    void (*menupont[3]) (void) = { beolvas, szamol, kilir };
    do {
        p = menu;
        while (*p)
            cout<<*<<endl;
        cin>>ch; cin.get();
        if (ch > '0' && ch < '4')
            menupont[ch-'1'] ();
        else if (ch != '0')
            cout<<'\a';
        while (ch != '0') ;
    }
}
```

Hasonlóan érdekes probléma az egy-szerű, kötött szerkezetű leírónyelven megírt programok (például NC, HP plotter stb.) gyors értelmezése. A parancsok nagy száma miatt a vezérlést végző függvény áttekinthetetlen és kusza lesz, ha a szokásos **if** vagy **switch** szerkezetet használjuk. Szép megoldáshoz jutunk azonban olyan mutatótömb kialakításával, melynek elemei a parancsoknak megfelelő függvényekre mutatnak.

Tegyük fel, hogy a bemutásra kerülő program által feldolgozott leírónyelv utasításai *„An paraméterek”* szerkezetűek! Az első betű ('A'-'Z') a parancs csoportját, míg a második számjegyekarakter ('0'-'9') a parancsot jelöli ki. Ez összesen 26x10=260 különböző-zo parancs megadását teszi lehetővé. A kétkarakteres parancsot szökőközzel elválasztva követik a parancs paraméterei.

A feldolgozó függvények egy mutatót kapnak argumentumként, amely az utasításban a *paraméterek* rész elejére mutat. Ezért e függvények prototípusa:

```
void parancs(char *);
```

A 260 parancsot kezelő függvény címének tárolására kétdimenziós tömböt használunk, így a parancs azonosítását az utasítás első két karaktere alapján elvégezhetjük:

```
void (* fvt[26][10]) (char *);
```

A feladat egy lehetséges megoldása az alábbiakban tanulmányozható:

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

void p0(char *);
void p1(char *);
void q0(char *);
void ures(char *);

// A feldolgozandó programrészt
char *prog[]={ "P0 100,300",
               "P1 Hello",
               "P2 100,LEFT",
               "P0 200,500",
               "Q1 500",
               "P1 Bye",
               "Q0 " };

int main()
{
    void (* fvt[26][10]) (char *);
    int i, j, n;

    for (i = 0; i < 26; i++) // inicializálás: ures
        for (j = 0; j < 10; j++)
            fvt[i][j] = ures;

    fvt['P','A']["0","0"] = p0; // inicializálás: p0, p1 és q0
    fvt['P','A']["1","0"] = p1;
    fvt['Q','A']["0","0"] = q0;

    // A példaprogram feldolgozása
    n=sizeof(prog)/sizeof(prog[0]);
    for (i = 0; i < n; i++)
        (*fvt[ prog[i][0]-'A' ][ prog[i][1]-'0' ])( &prog[i][3] );
}

// A parancsfüggvények definíciója
void ures(char * p) {
    cout<<"***** Ertvenytelen parancs *****"<<endl;
}

void p0(char * p) {
    int a,b;
    char ch;
    stringstream ist(p);
    ist>>a>>ch>>b;
    cout<<a<<" " + "<<b<<" = "<<a+b<<endl;
}
```

```

void p1(char * p) {
    cout<<p<<endl;
}

void q0(char * p) {
    cout<<"--- A feldolgozas vege ---<endl;
}

```

Ugyancsak érdemes egy pillantást vetni a feldolgozás eredményeként megjelölt képernyőtartalomra:

```

100 + 300 = 400
Hello
*** Brvenytelen parancs ***
200 + 500 = 700
*** Brvenytelen parancs ***
Bye
--- A feldolgozas vege ---

```

8.4.6 Alapértelmezés szerinti (default) argumentumok

A C++ függvények prototípusában bizonyos paraméterekhez ún. alapértelmezés szerinti értéket rendelhetünk. A fordító ezeket az értékeket használja fel a függvény hívásakor, ha az adott argumentum nem szerepel a hívási listában:

```
double defargfv(int a, double b=3.14, char c='K');
```

A példából is látható, hogy az alapértelmezés szerinti értékkel ellátott paraméterek jobbrol-balra haladva folytonosan helyezkednek el. A fenti függvény lehetséges hívásait felsorolva nézzük meg a paraméterek tényleges értékét!

Hívás	Paraméterek:	a	b	c
defargfv(10);		10	3.14	'K'
defargfv(10, 2.5);		10	2.5	'K'
defargfv(10, 2.5, 'X');		10	2.5	'X'

Nem megengedett hívási forma például a *defargfv(10, 'X')*. Ha valamely alapértelmezett argumentumot elhagyjuk, akkor az azt követő alapértelmezés szerinti értékkel ellátott argumentumokat is el kell hagynunk.

A következő példában használt *terület()* függvény háromszög területét határozza meg az oldalak ismeretében. (Derekcsözgü háromszög esetén a két befogó felhasználásával a terület egyszerűbben kiszámítható.) Amennyiben prototípust is használunk, akkor az alapértelmezés szerinti értékeket csak a prototípusban adhatjuk meg.

```

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

double terület(double a, double b, double c=0);

```



```
void main() {
    // Általános háromszög terület:
    cout << "\nÁltalános : " << terület(3,4,5);

    // Derékszögű háromszög terület:
    cout << "\nDerékszögű : " << terület(3,4);
}

double terület(double a, double b, double c) {
    if (c) {
        double s=(a+b+c)/2;
        return sqrt(s*(s-a)*(s-b)*(s-c));
    }
    else
        return (a*b/2);
}
```

Az alapértelmezés szerinti argumentumokkal ellátott prototípusok rugalmasabbá teszik a függvények használatát. Például, ha valamely függvényt sokszor hívunk egy-azon argumentumlistával, érdemes a gyakran használt argumentumokat alapértelmezés szerintivé tenni, és a hívást az argumentumok megadása nélkül elvégezni.

8.4.7 Változó hosszúságú argumentumlista

Bizonyos függvények esetén nem lehet pontosan megadni az argumentumok számát és típusát. Az ilyen függvények deklarációjában a paraméterlistát három pont zárja:

```
int printf(const char * formatumsztring, ... );
```

A három pont azt jelenti a fordítóprogram számára, hogy „még lehetnek további argumentumok”. A *printf()* függvény esetén legalább egy argumentumnak kell szerepelnie, amelyet tetszőleges számú további argumentum követhet:

```
printf("Hello C++ \n");
printf("A nevem: %s \n", nev);
printf("Az összeg: %d + %d = %d\n", a, b, c);
```

Felvetődik a kérdés, honnan tudja a *printf()*, hogy hány argumentumot kell feldolgoznia? A választ a *formatumsztring* adja, melyen végiglépkedve a formátum alapján dolgozza fel a *printf()* függvény a soron következő argumentumot.

Mivel az ilyen deklarációjú függvények hívásakor a fordító csak a „...” listaelemig képes az argumentumok típusát egyeztetni, ezért a további argumentumok esetén a hagyományos konverziókat hajtja végre. Más szavakkal, ebben az esetben a megadott argumentumok (esetleg konvertált) típusa szerint megy végbe az argumentumok átadása a függvénynek.

A C++ nyelv lehetővé teszi, hogy saját függvényeinkben is használjuk a három pontos – az ún. változó hosszúságú argumentumlistát. Ahhoz, hogy a paramétereket tartalmazó memóriaterületen megtaláljuk az átadott argumentumok értékét, legalább az első paramétert mindig meg kell adnunk.

A C++ szabvány tartalmaz néhány olyan makrót, amelyek segítségével a változó hosszúságú argumentumlista feldolgozásához nem kell ismernünk az adott számítógépes környezetet „lelkivilágát”. A *cstdlib* fejeletben deklarált, illetve definiált makrók a következők:

```

típus va_arg(va_list ap, típus);

Az argumentumlista következő elemét adja vissza.

void va_end(va_list ap);
„Nagyításkartás” az argumentumok feldolgozása után.

void va_start(va_list ap, fixparam);
Inicializálja az argumentumok eléréséhez használt mutatót.

```

A fenti makrók a *va_list* típusú mutatót használják az argumentumok eléréséhez.

Példaként tekintünk a tetszőleges számú **int** érték összegét kiszámító *osszeg()* függvényre, melynek hívásakor a számsort 0-val kell zárni! Az *atlag()* függvény segítségével adott darabszámú **double** érték átlagát számoljuk. A darabszámot a függvény első argumentumaként kell megadni.

```

// Egész számok összegzése 0-ig
int osszeg(int elso, ...) {
    va_list ap;
    int s = elso, ertek;
    va_start(ap, elso);
    while (ertek = va_arg(ap, int)) // az első argumentum átlépése
        s += ertek;
    va_end(ap);
    return s;
}

// Adott számú double érték átlagolása
double atlag(int n, ...) {
    va_list ap;
    double s = 0;
    va_start(ap, n);
    for (int i=0; i<n; i++)
        s += va_arg(ap, double); // az első argumentum átlépése
    va_end(ap);
    return s / n;
}

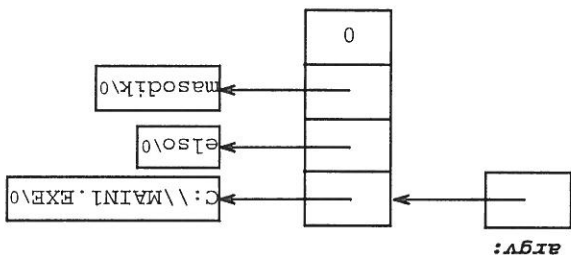
```

8.4.8 A main() függvény paraméterei és visszatérési értéke

A *main()* függvény különlegességét nemcsak az adja, hogy a program végrehajtása vele kezdődik, hanem az is, hogy nulla, egy vagy két paraméterre lehet:

```
int main ( )
int main ( int argc ) ( )
int main ( int argc, char *argv[] ) ( )
```

A paraméterek nevét tetszőlegesen megválaszthatjuk, azonban a program olvashatóságát jelentősen javítja a szabványos elnevezések használata. Az *argv* egy karakter-mutatókat tartalmazó tömbre (vektorra) mutat, az *argc* pedig a tömbben található stringek számát adja meg. (Az *argc* értéke legalább 1, mivel az *argv[0]* mindig a program nevét tartalmazó stringre hivatkozik.)



A *main()* visszatérési értékét, amely a szabvány ajánlása alapján *int* típusú, a *main()* függvényben belüli *return* utasításban, vagy a program tetszőleges pontján elhelyezett *exit()* könyvtári függvény argumentumában adhatjuk meg. Az *csdlib* fejeletlomány szabványos konstansokat is tartalmaz,

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

amelyeket kilépési kódként használva, a program sikeres, illetve sikertelen futását jelezhetjük. A *main()* függvényt bizonyos fordítóprogramok *void* típusúnak is elfogadják. Ekkor azonban a programunknak 0 vagy határozatlan lesz a kilépési kódja.

A *main()* sok mindenképpen különbözik más C++ függvényektől: nem deklarálhatjuk *static* vagy *inline* függvényként, és nem lehet túlterhelni. A szabvány ajánlása alapján a programon belülről nem hívható, és nem kérdezhető le a címe sem.

Az alábbi példaprogram (*MAIN1.cpp*) a segédprogramok írásánál jól felhasználható megoldást mutat be az indítási argumentumok helyes megadásának tesztelésére. A program csak akkor indul el, ha pontosan két argumentummal indítjuk. Ellenkező esetben hibajelzést és az indítási parancssor formáját bemutató üzenetet ír ki a képernyőre.

```

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

int i;
int i;
if (argc != 3) {
    cerr<<"Hiba paraméterezés!"<<endl;
    return EXIT_FAILURE;
}
cout<<"1. argumentum: "<<argv[1]<<endl;
cout<<"2. argumentum: "<<argv[2]<<endl;
cin.get();
return EXIT_SUCCESS;
}

```

A *main()* függvény argumentumainak megadása, illetve a visszatérési érték feldolgozása függ az operációs rendszertől. Az MS-DOS-ban az argumentumokat a programot indító parancssorból másolja be a futtató rendszer az *argv* által kijelölt tömbbe.

Ha a program neve *MAIN2*, akkor a hívási argumentumok megadására a

C:\CPP\MAIN2 első második 3. 4.

parancssor használható. Az argumentumok elválasztására a szóközt használjuk.

Az alábbi *MAIN2.cpp* program kírja a parancssor-argumentumok számát, megjeleníti az argumentum-karakter sorozatokat és az *envp* által mutatott sztringtömb tartalmát. Az *envp* a DOS-ban a környezeti változókat tartalmazó sztringtömböt jelöli, melyben az érvényes elemeket egy *NULL* érték zárja.

```

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

int i;
int i;
int main(int argc, char *argv[], char **envp) {
    cout<<"Az argumentumok száma: "<<argc<<endl<<endl;
    for (i=0; i<argc; i++)
        cout<<i<<" ". argumentum: \ "<<argv[i]<<" \ "<<endl;
    p = envp;
    while (*p)
        cout<<"Környezeti változó: \ "<<*p++<<" \ "<<endl;
    cin.get();
    return EXIT_SUCCESS;
}

```

A fenti parancssort használva, a program futásának eredménye:

```
Az argumentumok száma: 5
0. argumentum: "C:\MAIN2.EXE"
1. argumentum: "első"
2. argumentum: "masodik"
3. argumentum: "3."
4. argumentum: "4."
Környezeti változó: "COMSPEC=C:\COMMAND.COM"
Környezeti változó: "PATH=C:\DOS\HELP;"
Környezeti változó: "PROMPT=$P$G"
```

8.5 Rekurzív függvények használata

A matematikában lehetőség van bizonyos adatok és műveletek *rekurzív* definiálására. Minden *rekurzív* problémának létezik *iteratív* (ciklust használó) megoldása, amely általában sokkal nehezebben programozható, de hatékonysága miatt mégsem szabad megfélemedkezni róla! Klasszikus példaként tekintünk először a *Fibonacci* néven is ismert *Leonardo de Pisa* nyúl feladatát:

"Hány nyúlpatrunk lesz 3, 4, 5,..., n hónap múlva, ha egy nyulpat két hónapos kortól kezdve havonta egy-egy új párt hoz világra, feltéve, hogy az új párok is e törvény alapján szaporodnak, és mind életben maradnak."

A megoldást a *Fibonacci* számok sora tartalmazza (ha a 0 kezdőelemet figyelmen kívül hagyjuk):

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

A sor *n*-dik elemének meghatározására az alábbi rekurzív szabály szolgál:

$$\begin{aligned} a_0 &= 0 \\ a_1 &= 1 \\ a_n &= a_{n-1} + a_{n-2}, \quad n = 2, 3, 4, \dots \end{aligned}$$

A rekurzív szabály alapján elegáns megoldást kapunk, ha önmagát hívó rekurzív függvényt használunk. Ekkor gyakorlatilag a fenti összefüggést fogalmazzuk át C++ programmá:

```
unsigned long fibr( int n ) {
    if (n<2)
        return n;
    else
        return fibr(n-1) + fibr(n-2);
}
```

A rekurzív megoldás általában rövidebb és áttekinthetőbb, mint az iteratív megoldás, azonban a számítási idő és a memóriailágény jelentős növekedése miatt az esetek többségében mégis az iteratív megoldás használatát javasoljuk:

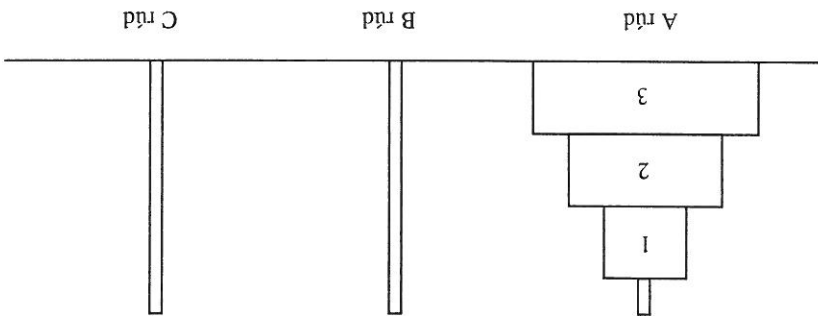
```

unsigned long fib( int n )
{
    unsigned long f0 = 0, f1 = 1, f2 = n;

    while (n-- > 1) {
        f2 = f0 + f1;
        f0 = f1;
        f1 = f2;
    }
    return f2;
}

```

A következő feladat megoldása a rekurzív alkalmazása nélkül igen bonyolult. A megoldandó probléma a Hanoi tornyai nevet viseli.



Adott három rúd: A, B és C. Az A rúdon induláskor N darab különböző átmérőjű lyukas korong helyezkedik el, az átmérők csökkenő sorrendjének megfelelően. A feladat a korongok átrakása a C rúdra, az alábbi szabályok figyelembevételével:

- a B rúd közbenso tárolásra használható,
- minden lépésben csak egy korong mozgatható,
- minden korong csak nálánál nagyobb átmérőjű korongra helyezhető.

(A feladat egy legenda alapján, amely szerint Hanoi közelében található kolostorban 64 aranykorongból álló tornyot raknak át a szerzetesek, a fenti szabályok betartásával, minden nap egyetlen korongot mozgatva. A legenda szerint akkor lesz vége a világnak, ha az átrakást befejezik.)

A feladat lehetséges megoldását az alábbi példaprogram tartalmazza:

```

#include <iostream>
using namespace std;

void Honnan_Hova_Mozgatas(char, char) ;
void Hanoi(int, char, char, char) ;

```

Önrekurzíóról akkor beszélünk, amikor egy függvény közvetlenül hívja önmagát. A fenti két példában ezt a megoldást használtuk. A rekurzív működés megértéséhez tekintünk a faktoriális számítását, amely rekurzív megoldással szintén hosszasan működik! Ebben az esetben is az iteratív (ciklusos) megoldás használata javasolt. A faktoriális-számítás rekurzív definíciója:

8.5.1.1 Önrekurzio

A rekurzív algoritmusok általában önrekurzóra vagy kölcsönös rekurzóra épülnek.

8.5.1 A rekurzív függvények csoportosítása

A korongok száma : 3				
Tedd a korongot a(z)	A	rudrol a(z)	C	rudra.
Tedd a korongot a(z)	A	rudrol a(z)	B	rudra.
Tedd a korongot a(z)	C	rudrol a(z)	A	rudra.
Tedd a korongot a(z)	B	rudrol a(z)	C	rudra.
Tedd a korongot a(z)	B	rudrol a(z)	A	rudra.
Tedd a korongot a(z)	A	rudrol a(z)	B	rudra.
Tedd a korongot a(z)	A	rudrol a(z)	C	rudra.

Három korong megadása esetén a program által javasolt megoldás:

```
void Hanoi(int n, char honnan, char mivel, char hova) {  
    if (n == 1)  
        Hanoi(n-1, honnan, hova, mivel);  
    else  
        Hanoi(n-1, honnan, hova, mivel);  
    Hanoi(n-1, mivel, honnan, hova);  
}
```

```
void Hanoi(char hova, char honnan, char mivel, char hova) {  
    cout<<"A korongok száma : ";  
    cin>>korongszam; cin.get();  
    cout<<endl;  
    Hanoi(korongszam, 'A', 'B', 'C');  
    cin.get();  
}
```

```
void Hanoi(char hova, char honnan, char mivel, char hova) {  
    cout<<"Tedd a korongot a(z)\t"<<innen<<"\t";  
    cout<<"rudrol a(z)\t"<<ide<<"\t rudra.\n";  
}
```

$$n! = \begin{cases} 1, & \text{ha } n = 0 \\ n \cdot (n-1)!, & \text{ha } n > 0 \end{cases}$$

Ennek alapján 4! kiszámításának lépései az alábbi szemléletes formában ábrázolhatók:

$$\begin{array}{rcl} 4! & = & 4 * 3! \\ & & 3 * 2! \\ & & 2 * 1! \\ & & 1 * 0! \\ & = & 4 * 3 * 2 * 1 * 1 = 24 \end{array}$$

A fenti számítási menetet megvalósító C++ függvény:

```
unsigned long long factr (int n)
{
    if (n == 0)
        return 1;
    else
        return n * factr (n-1);
}
```

A faktoriálissal összefüggő, rekurzíval megoldható probléma egy szó betűinek összes lehetséges módon történő összerakása (ismétlés nélküli permutáció). A megoldásban szereplő függvények (*main()*, *swapnm()* és *permut()*) közös globális változókat használnak.

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

// Globális változók a függvények közötti kommunikációhoz
char szo[8];
int nbetu, nperm;

void swapnm(char, char);
void permut(int);

void main()
{
    cout<<"Kérek egy max. 7 betus szot: ";
    cin.getline(szo,8);
    nbetu = strlen(szo);
    nperm = 1;
    if (nbetu > 0 && nbetu < 8)
        permut(nbetu-1);
    cin.get();
}
```



```

    typedef void fv(int);
    fv fv_a, fv_b;
    // A függvények típusa
    void fv_a(int i) {
        //
        if (i > 0) fv_b(i / 2); // Az fv_b hívása
    }
    void fv_b(int j) {
        //
        if (j > 0) fv_a(j - 1); // Az fv_a hívása
    }

```

Kölcsönös rekurzióról akkor beszélünk, amikor egy függvény az általa hívott másik függvényből hívódik meg újra. Kölcsönös rekurzió esetén különösen oda kell figyelni a rekurziót leállító feltétel megadására. Az ilyen megoldásokban szükséges a függvények prototípusának megadása:

8.5.1.2 Kölcsönös rekurzió

```

1 - ++C
2 - ++C
3 - ++C
4 - ++C
5 - ++C
6 - ++C

```

Kerek egy max. 7 betűs szót: C++

A program működése során sorszámozza az előállított permutációkat:

```

// Két karakter felcserélése a szóban
void swapnm(char n, char m) {
    char z;
    z = szo[n];
    szo[n] = szo[m];
    szo[m] = z;
}

void permuto(int k) {
    if (k)
        swapnm(i, k);
    for (int i=0; i<=k; i++) {
        permuto(i);
        cout<<"\n\t"<<permuto++<<" - "<<szo;
        swapnm(k, i);
    }
}

```

8.6 Inline függvények

A C++ előfordító (preprocesszor) *#define* direktívájának használata során olyan hibákat vihetünk be a programunkba, amelyek a forráskód áttanulmányozásával nem derülnek ki (kifejezés megadása makróban, vagy a léptető operátorok használata a makró argumentumában). A C++-ban javasolt a *#define* használatainak korlátozása, hiszen az esetek többségében a *const* és *inline* definíciók kiválják azt.

Az *inline* („soron belüli”) függvényekkel a *#define* makrókat helyettesíthetjük. Az *inline* függvény esetén a függvény törzset képező kód helyettesítődik be a függvény-hívás helyére („kódmakró”). Például, a *MAX* makró makró helyett egész értékek esetén a *max()* *inline* függvény használata javasolt.

```
#define MAX(x,y) (x)>(y)?(x):(y)

inline int max(int x, int y)
{
    return (x>y?x:y);
}
```

Nézzük a hivatkozást a makróra és az *inline* függvényre!

```
void main()
{
    int a;
    a=MAX(4,26); // az előfordító a=(4)>(26)?(4):(26); utasításá-
                  // alakítja át.
    a=max(4,26); // a fordító a függvény törzset képező kódot
                  // helyettesíti be az utasításba: a=x>y?x:y;
}
```

Tudnunk kell azonban, hogy az *inline* definíció csak javaslat a fordító számára, amelyet az bizonyos feltételek esetén (de nem mindig!) figyelembe is vesz. Alapában kíséreltet, gyakran hívott függvények esetén ajánlott alkalmazni ezt a megoldást.

Az *inline* függvények használatának előnye, hogy a függvényhíváskor az argumentumok feldolgozása teljes körű típusellenőrzés mellett megy végbe. Talán ez egyben a hátránya is, hiszen a *MAX* makró tetszőleges aritmetikai típus esetén használható, míg a *max()* függvény csak *int* típusú argumentummal hívható. Ezen probléma kiküszöbölésében a C++ egy nagyon fontos mechanizmusa, a függvénynevek átdefiníálása (túlterhelése, *overloading*) segít.

8.7 Függvénynevek átdefinálása (overloading)

A C++-ban több függvényt is definiálhatunk ugyanazzal a névvel, ha a definiált függvények paraméterlistája („kézjegye”) eltér egymástól. Az így definiált függvények közül az éppen szükségességet a fordító választja ki a hívási argumentumok száma és típusa alapján.

Az alábbi példában a *sumall()* függvénynek két átdefiníált formája létezik, az *int* és a *double* típusú tömbök elemösszegének meghatározására:

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

int sumall(int a[], int n)
{
    int sum=0;
    for (int i=0; i<n; i++)
        sum+=a[i];
    return sum;
}

double sumall(double a[], int n)
{
    double sum=0;
    for (int i=0; i<n; i++)
        sum+=a[i];
    return sum;
}

void main()
{
    int a[]={1,1,2,3,5,8,13};
    const int n1=sizeof(a)/sizeof(a[0]);
    cout << "\nAz int tömb elemösszege: "<<sumall(a1,n1);

    double ad[]={1.2,2.3,3.4,4.5,5.6};
    const int nd=sizeof(ad)/sizeof(ad[0]);
    cout << "\nA double tömb elemösszege: "<<sumall(ad,nd);
}
```

A fordító az első híváskor *sumall(int *, const int)*, ezért például *unsigned* és *float* tömbök esetén fordítási hibát kapunk, hiszen C++-ban a mutatók automatikus konverziója erősen korlátozott.

Láthatjuk, hogy a fenti két függvény csak a tömbök típusában tér el. További típusok bevezetése esetén szövegszerkesztési feladatná válik az újabb átdefiníalt függvények megírása (blokkmásolás, szövegcseré). Ennek elkerülése érdekében az átdefiníalt függvényváltozatok előállítására is rábízható a fordítóprogramra függvénynyilván

(*template*) definiálásával. A megfelelő függvényválasztás kiválasztásában a függvény visszatérési típusa nem játszik szerepet.

Átdefiniált névű függvények esetén a megfelelő függvény kiválasztása több lépésben megy végbe:

1. Keresés a teljes (egzak) típusegyezőség feltételének felhasználásával. (A teljes típusegyezőség esetén a **float** nem egyezik meg a **double**, és a **char** nem egyezik meg az **int** típussal.)

2. Keresés az ún. triviális típuskonverziók végrehajtásával. A C++ nyelvben az alábbi konverziókat tekintjük triviálisnak. (Az utolsó konverzió a függvény neve és függvényre mutató pointer közötti automatikus konverziót jelöli.)

<i>Típusról</i>	<i>Típusra</i>
típus	típus&
típus	típus
típus[]	típus*
típus	const típus
típus	volatile típus
f(v(argumentumok))	(*fv)(argumentumok)

3. Keresés az egész típusok közötti konverziók (**bool** → **int**, **char** → **int**, **short** → **int**, **unsigned char** → **int**, **unsigned short** → **int**), illetve a **float** → **double** átalakítás felhasználásával.

4. Keresés a szabványos típuskonverziók alkalmazásával. Ezek a konverziók alapvetően a szokásos aritmetikai (**int** → **double**, **signed** → **unsigned**), és a mutatókonverziókat jelentik. A pointerkonverziókat az alábbiakban foglaltuk össze. (A referenciátípusok konverziója a mutatótípusok konverziójának megfelelően megy végbe.)

<i>Típusról</i>	<i>Típusra</i>
tetszőleges mutatótípus	void *
származtatott osztály mutatója	az alaposztályra mutató pointer
0 konstans	NULL pointer.

5. A típusegyezőség keresése ideiglenes objektum létrehozásával.
6. Keresés a felhasználó által definiált konverziók alkalmazásával.

A függvényekhez hasonló módon használható a műveletek átdefiniálásának mechanizmusa (*operator overloading*). Az operátorokat azonban csak felhasználó által definiált típusokkal (**struct**, **union**, **class**) lehet átdefiniálni, ezért ezzel a lehetőséggel könyvünk következő részében foglalkozunk részletesen.

8.8 Általánosított függvények (template)

A függvények áttekinthetőségénél láttuk, hogy sok esetben ugyanazt a függvényfelépítést használjuk a túlterhelt függvényválasztásokban, csupán néhány típust cserélünk le. Ilyen esetekben a fordító segíthet nekünk az áttekinthetőség érdekében, hogy mely függvények esetén, mely típusokat kell lecserélnie, vagyis el kell készítenünk egy általánosított függvénymintát (sablon, *template*).

A függvénysablon előállításakor érdemes kiindulnunk egy működő függvényből, amelyben a lecserélendő típusokat általános típusokkal (*TIPUS*) helyettesítjük. Ezek után közöljük a fordítóval, hogy mely típusokat kell lecserélni a függvény-mintában (*template <class TIPUS>*). (Az itt szereplő *class* kulcsszónak nincs semmi köze az osztályokhoz.)

```
template<class TIPUS>
TIPUS sumall(TIPUS a[], int n)
{
    TIPUS sum=0;
    for (int i=0; i<n; i++)
        sum+=a[i];
    return sum;
}
```

Az elkészült függvénysablonból a fordító állítja elő a szükséges függvényválasztásokat, amikor először egy hívással találkozunk. Az előző alfejezet példája sokkal áttekinthetőbbé válik *template* felhasználásával:

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

template<class TIPUS>
TIPUS sumall(TIPUS a[], int n)
{
    TIPUS sum=0;
    for (int i=0; i<n; i++)
        sum+=a[i];
    return sum;
}

void main()
{
    int a1[]={1,1,2,3,5,8,13};
    const int n1=sizeof(a1) / sizeof(a1[0]);
    cout << "n1 az int  OMB elemosszege: "<<sumall(a1,n1);

    double ad[]={1.2,2.3,3.4,4.5,5.6};
    const int nd=sizeof(ad) / sizeof(ad[0]);
    cout << "nA doubleOMB elemosszege: "<<sumall(ad,nd);
}
```

```

float af[]={3, 2, 4, 5};
const int nf=sizeof(af) / sizeof(af[0]);
cout << "na float tomb elemszage: "<<sumall(af,nf);

long a1[]={1223, 17122399, 233594};
const int n1=sizeof(a1) / sizeof(a1[0]);
cout << "na long tomb elemszage: "<<sumall(a1,n1);
cin.get();
}

```

A függvényablonban természetesen több típus helyettesítése is megoldható, mint ahogy azt a következő példában láthatjuk:

```

template <class T1, class T2>
inline T1 max(T1 a, T2 b)
{
    return (a>b? a : b);
}

int main()
{
    cout<<max(5,4)<<endl;
    cout<<max(5.6,4)<<endl;
    cout<<max('A',66L)<<endl;
    // int max(int, int);
    // int max(double, int);
    // char max(char, long);
}

```

Az ANSI/ISO C++ nyelven a függvényablonokat paraméterezhetjük is. Ebben az esetben a függvény hívásakor a függvény neve mögött meg kell adnunk a *template* argumentumait. Az alábbi példában a *Kiiras()* általánosított függvény hívásakor a típuson túlmenően a mezősszélességet és a pontosságot is beállítjuk.

```

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

template <class T, int w, int p>
void Kiiras(T a)
{
    cout.width(w);
    cout.precision(p);
    cout<<a<<endl;
}

int main()
{
    Kiiras<int, 10, 8>(1234567);
    Kiiras<char, 12, 10>("C++ nyelv");
    Kiiras<double, 12, 7>((1223.3445));
    cin.get();
}

```

8.9 Típusmegőrző szerkesztés (type-safe linking)

A szabványos C++ nyelv támogatja a típusmegőrző szerkesztést, amely segíti a hibás argumentummal történő függvényhívások kiszűrését a szerkesztés folyamán. A típus-megőrző szerkesztés mechanizmusának jobb megértése érdekében nézzünk néhány *Borland C++ Builder* rendszerben használt névképzési példát!

Prototípus	Tárgykódbeli név
<code>double sumall(double a[], int n);</code>	<code>@sumall\$qpdl</code>
<code>int sumall(int* a, int n);</code>	<code>@sumall\$qpdl</code>
<code>int fvl(int& a, bool n);</code>	<code>@fvl\$qrlo</code>
<code>extern "C" int fvl(int& a, bool n)</code>	<code>extern "C" int fvl</code>

A C++ fordítók fent bemutatott névképzése teszi lehetővé a függvénynevek átdefiníciá-lását (ülterhelését), hisz ekkor a függvény tényleges nevének kialakítása függ a függ-vény paraméterezésétől. Megjegyezzük, hogy valamely osztály tagfüggvényeinek ne-vei az osztály nevét is tartalmazzzák.

A függvénynevek ilyen formán történő képzése természetesen megnehezíti más nyelv-ven (például C) írt modulok, könyvtárak C++ nyelvből való hasznosítását. A probléma leküzdésére a C++ nyelv tartalmazza az **extern "C"** típusmódosítót. Ha például C nyelven megírt *sin()* függvényt szeretnénk használni a C++ programból, szükséges az

```
extern "C" double sin(double a);
```

deklaráció megadása. Ennek hatására a C++ fordító a szokásos C-névképzést használ-ja a *sin()* függvényre. Ebből persze az is következik, hogy a nem C++ függvényeket nem lehet átdefinálni (ülterhelni)! Ha több C függvényt kívánunk meghívni, akkor az **extern "C"** deklaráció csoportos formáját használhatjuk:

```
extern "C" {
    double sin(double a);
    double cos(double a);
}
```

Ha a C-könyvtár függvényeit külön deklarációs állomány tartalmazza, akkor a követ-kező megoldás áll rendelkezésünkre:

```
extern "C" {
    #include <stdio.h>
}
```

8.10 Esettanulmány: C++ deklarációk értelmezése és készítése

A C++ nyelv deklarációinak formája (szintaxisa) jelentősen különbözik más programozási nyelvekben használt deklarációktól. Egy-egy bonyolultabb C++ deklaráció pontos jelentése még a gyakorlott programozóknak sem azonnal érthető. Ezért ebben a fejezetben a C++ típusok összefoglalásaként kitérünk arra, hogyan kell értelmezni, illetve elkészíteni a C++ deklarációkat.

A C++ nyelv ugyanazt az operátor- és szimbólumkészletet használja a deklarációkban, mint a kifejezésekben szereplő azonosítók esetén. Az alábbi példában egy egész típusú változót (x), egy mutatót (px) és egy referenciát (rx) hozunk létre:

```
int x;
int *px;
int &rx=x;
```

A $*px$ deklarációnak ugyanaz a formája, mint amikor kifejezésben a mutatót egész változót jelöli:

```
x = *px;
```

A példában bemutatott szimmetria jól használható a kifejezések és a deklarátorok típusának megállapításakor: a $*px$ olyan egész változó, amelyre a px pointer mutat.

A C++ szabvány a deklaráció és a definíció fogalmak mellett a deklarátor fogalmát is használja. Deklátor alatt a deklarálandó objektumot és a hozzákapcsolt módosítókat (mutató $*$, referencia $\&$, tömb $[]$ vagy függvény $()$) értjük.

8.10.1 C++ deklarációk értelmezése

A fentnél bonyolultabb deklarációk értelmezéséhez azonban további magyarázatok szükségesek. Fontos megjegyeznünk, hogy a C++ deklarátorokban használt szimbólumok valójában C++ operátorok, amelyekre érvényes a precedencia és az asszociativitás szabálya. A precedencia szerint az alábbi két csoportból kerülnek ki a dekláratórokban használt operátorok:

1. A függvényhívás operátora $()$ és a tömbindexelés operátora $[]$ azonos precedenciával rendelkeznek. Ezen operátorok csoportosítása balról-jobbra történik.
2. Az indirekt hivatkozás operátora $*$ és a címe operátor $\&$ szintén azonos precedenciával rendelkeznek. Több operátor használata esetén a csoportosítás jobbról-balra történik.

Tekintsük a következő példát!

```
int *x[3];
```


A kérdés, hogy ez most egésze mutató pointerek tömbje, vagy pedig egészeket tartalmazó tömbre mutató pointer? A választás érdekében a kifejezést szétbontjuk, méghozzá a precedencia- és az asszociativitás-szabályokkal ellentétes irányban:

1. `*[3]` egy egész,
2. `*[3]` egésze mutató pointer,
3. `x` egésze mutató pointerek 3-elemű tömbje.

A legbonyolultabb deklarátorok és kifejezések értelmezése gyorsan elvégezhető ezzel a szétbontásos módszerrel. (A csillagot azért vettük le először, mivel kisebb a precedencia az indexelés operátorénál.)

A szétbontás általános szabályát is megfogalmazhatjuk. Először sorban le vesszük a legkisebb precedenciájú operátorokat. Ezt követően, ha a megmaradt operátorok precedenciája megegyezik, akkor nézzük az asszociativitást. A balról-jobbra csoportosított operátorok esetén először a jobbszélű, míg a jobbról-balra csoportosított operátorok esetén először a baliszélű operátort vesszük le. Minden egyes lépés után megfogalmazzuk, hogy mi az, amit kapunk. A lebontásnál természetesen a zárójelket is figyelembe kell venni.

Ezzel a szabállyal könnyedén értelmezhetjük a leggyakrabban használt deklarációkat. Sajnos a magyar nyelv nem teszi lehetővé, hogy a típusokat az angol nyelvben használt egyszerűséggel írjuk fel. Nézzünk néhány példát!

```
char **argv      karakterre mutató pointerre mutató pointer,
int *p;          g referencia egy egésze mutató mutatóra,
int * & q = p;
```

```
int a[5]          egészeket tartalmazó 5-elemű tömb,
int (*pa)[5]      egészeket tartalmazó 5-elemű tömbre mutató pointer,
int *pv[5]        egésze mutató pointereket tartalmazó 5-elemű vektor,
```

```
char *fv(int)     függvény, amely int argumentumot kap, és karakterre mutató
                  pointerrel tér vissza,
char & rfv(int)   függvény, amely int argumentumot kap, és karakter-referen-
                  ciával tér vissza,
char (*pfv)(int)  egy olyan függvényre mutató pointer, amely int argumentumot
                  kap karaktert ad vissza.
char * (xrfv)(int) = fv;  referencia egy olyan függvényre, amely int argumentumot kap,
                          és karakterre mutató pointerrel tér vissza.
```

Az Olvasó megnyugtatósára közöljük, hogy ezeknél bonyolultabb deklarációk nem túl gyakran fordulnak elő a C++ programokban.

Tekintünk az alábbi egyszerűnek nem nevezhető deklarációkat, és értelmezzük azokat! A deklarációk értelmezéséhez használjuk a lebontás elvét!

```
int (* (*x()) [6]) ();
```

1. (* (*x()) [6]) ()
egy egész,
2. (* (*x()) [6])
függvény, amely egészet ad vissza,
3. (*x()) [6]
egészet visszaadó függvényre mutató pointer,
4. *x()
egészet visszaadó függvényre mutató pointerek 6-elemű tömbje,
5. x()
mutató az egészet visszaadó függvényre mutató pointerek 6-elemű tömbjére,
6. x
függvény, amely egészet visszaadó függvényre mutató pointerek 6-elemű tömbjére mutató pointerrel tér vissza.

Az előzőhöz formailag hasonló, azonban lényegileg különböző az alábbi deklaráció:

```
int (* (*x) [6]) ();
```

1. (* (*x) [6]) ()
egy egész,
2. (* (*x) [6]) ()
egésze mutató pointer,
3. * (*x) [6]
egésze mutató pointer visszaadó függvény,
4. (*x) [6]
egésze mutató pointer visszaadó függvényre mutató pointer,
5. *x
egésze mutató pointer visszaadó függvényre mutató pointerek 6-elemű tömbje,
6. y
egésze mutató pointer visszaadó függvényre mutató pointerek 6-elemű tömbjére mutató pointer.

A fenti két függvénydeklaráció után nézzünk egy tömbdeklarációt:

```
char (* (*z[2]) ()) [5];
```

1. (* (*z[2]) ()) [5]
egy karakter,
2. * (*z[2]) ()
5-elemű karaktertömb,
3. (*z[2]) ()
5-elemű karaktertömbre mutató pointer,
4. *z[2]
5-elemű karaktertömbre mutató pointer visszaadó függvény,
5. z[2]
5-elemű karaktertömbre mutató pointer visszaadó függvényre mutató pointer,
6. z
5-elemű karaktertömbre mutató pointer visszaadó függvényre mutató pointereket tartalmazó 2-elemű tömb.

8.10.2 C++ deklarációk készítése

A program írása szempontjából a saját deklarációk megfelelő kialakítására kell a hangsúlyt helyezni. A deklaráció felépítése az előző alfejezetben alkalmasított lebontási módszerrel ellentétes művellet, így az ott alkalmasított szabályokat is fordítva kell alkalmaznunk. Azonban az igazán megbízható és olvasható deklarációk csak a **typedef** típusdeklaráció segítségével készíthetők.

A példaként elkészített deklaráció elég bonyolult, azonban ennek megértése után az egyszerűbb (és az összetettebb) deklarációk írása már nem jelenthet gondot.

Készítsük el annak a függvénynek a deklarációját, amely karakterre mutató pointerek 5-elemű tömbjére mutató pointerrel tér vissza!

```
f
függvény, amely karakterre mutató pointerek 5-elemű tömbjére
mutató pointerrel tér vissza,
f()
karakterre mutató pointerek 5-elemű tömbjére mutató pointer,
*f()
(*f()) [5]
karakterre mutató pointer,
karakter,
char * (*f()) [5]; a kész deklaráció!
```

A megoldást a **typedef** felhasználásával, a deklarációk elemi lépéseiből állítjuk elő:

```
// karakterre mutató pointer típusa
typedef char *cp;
// karakterre mutató pointerek 5-elemű tömbjének típusa
typedef cp v5cp[5];
// karakterre mutató pointerek 5-elemű tömbjére mutató
// pointer típusa
typedef v5cp *pv5cp;
// olyan függvény típusa, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
typedef pv5cp fpv5cp();
// g olyan függvény, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
pv5cp g();
// az f olyan függvény, amely karakterre mutató pointerek
// 5-elemű tömbjére mutató pointerrel tér vissza
char * (*f()) [5];
// A deklarációk ekvivalenciájának ellenőrzése
fpv5pc *pf=f;
fpv5pc *pg=g;
```


9. Névterek és tárolási osztályok

Ahhoz, hogy igazán megértsük a C++ programok működését, fontos ismernünk azokat a szabályokat, amelyek meghatározzák, hogy a programon belül miként lehet használni a különböző változókat és függvényeket. Ahhoz, hogy a C++ fordító korrekt kapcsolatot tudjon kialakítani az azonosítók és a tárolók között, szükséges, hogy minden azonosító rendelkezzen legalább két jellemzővel – típussal és tárolási osztállyal.

A *típus*, amellyel a 4. fejezetben részletesen foglalkoztunk, kijelöli a változó számára lefoglalt memóriaterület méretét, illetve a benne tárolt adat értelmezését.

A *tárolási osztály* egyrészt meghatározza, hogy a változó hol jön létre a memóriában (regiszterben, statikus vagy dinamikus tárterületen), másrészt pedig definiálja a változó élettartamát. A tárolási osztályt (*auto*, *register*, *static*, *extern*) megadhatjuk a változó-definíciókban, illetve ha onnan hiányzik, akkor maga a fordító határozza meg azt, a definíció a programszövegben való elhelyezkedése alapján. Mielőtt rátérnénk a tárolási osztályok részletes tárgyalására, szükséges tisztáoznunk néhány fogalom jelentését:

- élettartam (*lifetime*),
- láthatóság (*visibility*),
- érvényességi tartomány (hatókör, *scope*),
- kapcsolódás (*linkage*),
- névterület (*namespace*).

9.1 Az azonosítók élettartama

Az élettartam (*lifetime*) a program végrehajtásának olyan időszak, amelyben az adott változó vagy függvény létezik. Az élettartam és a tárolási osztály szoros kapcsolatban állnak. Az élettartam szempontjából az azonosítókát három csoportra oszthatjuk: globális (statikus), lokális (automatikus) és dinamikus élettartamú nevek.

Statikus (globális) élettartam

Azok az azonosítók, amelyeket a **static** vagy **extern** tárolási osztállyal rendelkező statikus élettartamúak. Például, minden függvény és minden külső (a függvényekkel azonos) szinten definiált azonosító globális élettartamú.

A statikus élettartamú (globális) azonosító számára kijelölt memóriaterület (és a benne tárolt adatok) a program futásának teljes időtartama alatt megmarad. A globális változók inicializálása egyetlen egyszer – a program indításakor – megy végbe.

Felhívjuk a figyelmet arra, hogy az azonosító élettartama és elérhetősége nem azonos fogalmak. Vannak esetek, amikor a program végrehajtásának teljes ideje alatt létező globális azonosító nem érhető el a program tetszőleges pontjáról (ilyenek például a statikus lokális vagy a statikus globális változók.) Az azonosítók elérhetőségét az élettartam és a láthatóság együtt határozzák meg.

Automatikus (lokális) élettartam

A függvényen (blokkon) belül a **static** tárolási osztály nélkül definiált azonosítók automatikus élettartammal rendelkeznek. Szintén automatikus élettartammal rendelkeznek a függvényen belül deklarált, kapcsolódás nélküli azonosítók és a függvények paraméterei.

Az automatikus élettartamú (lokális) azonosító memóriaterületére (és a benne tárolt adatok) csak abban a blokkban létezik, amelyben az azonosítót definiáltuk. A lokális azonosítóhoz a blokkba történő minden egyes belépéskor új terület kerül lefoglalásra, ami blokkból való kilépés során megszűnik (tartalma elvesz). Következésképpen, ha a lokális változót kezdőértékekkel látjuk el, akkor az inicializálás mindig újra megtörténik, amikor a változó létrejön.

Dinamikus élettartam

Dinamikus élettartammal rendelkeznek azok a memóriaterületek, amelyeket a **new** operátorral lefoglalunk, illetve a **delete** operátorral felszabadítunk.

9.2 Ervényességi tartomány és a láthatóság

A tárolási osztállyal ugyancsak szoros kapcsolatban álló fogalom az azonosítók láthatósága (*visibility*), illetve érvényességi tartománya (hatóköre, *scope*). Az azonosító csak a hatókörén belül látható. Az érvényességi tartomány a program azon részét jelöli ki, amelynek határain belül az adott azonosítót felhasználhatjuk a tároló elérésére. Az érvényességi tartományok több fajtáját különböztetjük meg: blokkszintű (lokális), fájlszintű (globális), függvény-, prototípus- vagy osztályszintű.

- A **blokkszintű** érvényességi tartománnyal rendelkező azonosító csak abban a blokkban látható, amelyikben deklaráltuk. Amikor a program eléri a blokkot záró '}' zárójelét, az azonosító többé nem lesz elérhető.

- A **fájlszintű** érvényességi tartománnyal rendelkező azonosító abban a fordítási egységben látható, amely a deklarációját tartalmazza. Csak azok az azonosítók rendelkeznek fájlszintű érvényességi tartománnyal, amelyek globálisan, vagyis minden függvényen kívül deklaráltak. Amennyiben valamely függvényből olyan globális változót kívánunk használni, amelynek definícióját egy másik állomány tartalmazza, akkor az **extern** tárolási osztály felhasználásával kell az azonosítót deklarálni (külső azonosító).

- Az erőnyességi tartomány fogalma hasonló a kapcsolódás (*linkage*) fogalmához, azonban nem teljesen egyezik meg azzal. Azonos nevekkkel különböző erőnyességi tartományokban különböző azonosítókat jelölhetünk. Azonban a különböző erőnyességi tartományban deklarált, illetve az azonos erőnyességi tartományban egyenl több-ször deklarált azonosítók a kapcsolódás mechanizmusának felhasználásával ugyanarra a változóra vagy függvényre hivatkozhatnak. A kapcsolódás kijelöli azt a program-rész, amelyben az adott azonosítóra hivatkozhatunk (láthatóság). Háromféle kapcsolódást különböztetünk meg: belső, külső, és amikor nincs kapcsolódás.
- A *belső kapcsolódású* azonosítók csak egyetlen fordítási egységen (modul-ban) belül ismertek. Ha a fájlszintű erőnyességi tartománnyal rendelkező változó- vagy függvényazonosítók deklarációját tartalmazza a **static** kulcsszó, akkor belső kapcsolódású azonosító jön létre, amely más fordítási egységből nem érhető el. (Ellenkező esetben külső kapcsolódással rendelkező globális azonosítók.) A C++ szabvány a **static** kulcsszó ilyen módon történő felhasználását *elavultnak* nyilvánította, helyette a *névtelen névtűlelek* használatát javasolja belső kapcsolódású azonosítók kialakítására.
- A *külső kapcsolódású* azonosítók több fordítási egységen (modulban) is ismertek. Azok a globális változó- vagy függvényazonosítók, amelyek deklarációjában nem szerepel tárolási osztály, vagy az **extern** tárolási osztály szerepel, külső kapcsolódásúak.
- A *kapcsolódás nélküli* azonosítók – – – valamely függvény vagy blokk helyi (lo-kális) azonosítói – nem rendelkeznek állandó memóriaterülettel. C++ nyelven az alábbi azonosítók nem rendelkeznek kapcsolódással:
 - minden olyan azonosító, amely nem jelöl változót vagy függvényt (például az **enum** konstansok, a címkek stb.),
 - a függvényparaméterek,
 - olyan blokkszintű hatókörbeli változó-azonosítók, melyek deklarációjában nem szerepel az **extern** kulcsszó.

9.3 A kapcsolódás

- Az egyetlen *függvény szintű* erőnyességi tartománnyal rendelkező C++ nyelv-*vi* erőnyességi azonosítók, amelyek csak a függvényen belül látható.
- Azok a paraméterazonosítók, amelyeket a függvények prototípusában megadunk (használatuk nem kötelező), csak a prototípust lezáró pontosvesszőig – a *prototípus szintű* hatókörben – láthatók.
- Az osztályokon belül definiált **private** és **protected** adattagok, illetve tag-függvények csak az osztály más tagfüggvényeiből (valamint a „barát” függvényekből) érhetőek el – *osztály szintű* láthatóság.

9.4 Névtérületek

A fordító a programban használt neveket (azok felhasználási módjától függetlenül) különböző területeken (névtérület – *namespace*) tárolja. Valamely névtérületen belül tárolt neveknél egyedieknek kell lenniük, azonban a különböző névtérületeken azonos neveket is szerepeltethetnek. Két azonos név, amelyek azonos értenységi tartományban helyezkednek el, de mincsemek azonos névtérületen, különböző azonosítókát jelölnek. A C++ fordító az alábbi névtérületeket különbözteti meg:

Utasításcímkek:

A nével ellátott utasításcímke, amelyet mindig kettőspont ':' követ, része az utasításnak. Nem szükséges, hogy a különböző függvényekben használt címkek eltérőek legyenek.

Struktúrák, osztályok és uniók tagjai

Az adattagok és a tagfüggvények nevei az adott felhasználói típushoz tartozó névtérületen helyezkednek el. Ezért valamely tag nevét egyidejűleg több felhasználói típusban is felhasználhatjuk. A tagok elérése a pont (.) vagy a nyíl (->) operátor megadásával lehetséges.

Közösleges nevek

Minden más név – a változók, a függvények, a paraméterek, a lokális változók és az **enum** konstansok nevei – közös névtérületen tárolódnak. A változó-azonosítók egy-másba ágyazott láthatósággal rendelkeznek, ami azt jelenti, hogy egy új blokkban újradefiniálhatók.

Típusnevek

A típusnevet azonos értenységi tartományon belül nem lehet azonosító nevéként használni.

Globális névtér

A globális névtér tartalmazza a fájlszinten definiált változóinkat és függvényeinket. A ISO/ANSI szabvány előtti C++ változatokban ugyancsak a globális névtérben helyezkedtek el a könyvtári függvények és osztályok. (A globális névtérület azonosítóra a hatókör operátorral hivatkozhatunk, például ::*nata*.)

9.4.1 Saját névtérületek kialakítása és használata

A szabványos C++ lehetővé teszi, hogy a névtérkezeletét saját kezünkbe vegyük. A névtérületek kialakításával a fájlszintű láthatóságot programmszintűvé terjeszthetjük ki azzal, hogy a globális névtér mellett saját, nével ellátott névtérületeket alakítunk ki.

A névtérület kijelölése tetszőleges forrásállományban (.cpp, .h) megadható, a fordító az azonos néven szereplő definíciókat egyesíti. A kijelölést a **namespace** kulcsszó felhasználásával végezzük:

```
namespace nevter {
    deklarációk és definíciók
}
```

Az alábbi példában szereplő névtér különböző definíciókat és deklarációkat tartalmaz:

```
namespace Pelda {
    int a=12;
    extern double d;
    double sqr(double x) {
        return x*x;
    }
    int mod(int a, int b);
    // függvény-deklaráció
}
```

Amennyiben a névtérületen belül deklarációk szerepelnek, az azokhoz tartozó definíciókat a névtérn kívül is megadhatjuk, használva a hatókör operátort:

```
int Pelda::mod(int a, int b)
{
    return a*b;
}
double Pelda::d=100;
```

A névtérületen megadott azonosítók minden olyan modulból elérhetők, amelybe beépítjük a névtér deklarációkat tartalmazó változatait (például fejállományból), esetenként:

```
namespace Pelda{
    extern int a;
    extern double d;
    double sqr(const double x);
    int mod(int a, int b);
}
```

Ezek után a hatókör (::) operátorsegítségével egyenként elérhetjük a neveket:

```
Pelda::d = Pelda::sqr(12.23);
Pelda::a=7430;
int x = Pelda::mod(Pelda::a, 12);
```

Természetesen sokkal kényelmesebb megoldáshoz jutunk, ha az összes nevet elérhetővé tesszük a magunk számára a `using namespace` direktívával:

```
using namespace Pelda;
d = sqr(12.23);
a=7430;
int x = mod(a, 12);
```

A `using` direktíva felhasználásával a névtér elemei közül csak a számunkra szükségeseket írjuk el:

```
using Pelda::d;
using Pelda::sqr;
d = sqr(12.23);
using Pelda::a;
a=7430;
using Pelda::mod;
int x = mod(a, 12);
```

A névtérre további lehetőségeinek bemutatása messze meghaladja fejezetünk méretét. A névtérülteket egymásba lehet skatulyázní, a `static` tárolási osztály (`extern` helyén való) használatát ún. névtelen névtérrel ajánlott felváltani, a névtérrekehez álnévtérreket definiálhatunk stb. Az elmondottak bemutatására nézzünk egy saját könyvtár kialakítását bemutató példaprogramot!

```
// -----
// lib.h állomány a szükségseges deklarációkat teszi a névtérbe
namespace lib {
    #include <math.h>
    #include <stdlib.h>
}

namespace lib {
    void rendez(int *v, int n, bool nov=true);
    void kilir(const int *v, int n);
}

// -----
// lib.cpp fájl a könyvtár elemeit definiálja
#include <iostream>
namespace rendszer = std; // álnév létrehozása
#include "lib.h"
using namespace lib;
```

```
// Névtelen névtér a modulszintű definíciókhoz
namespace {
    void csere(int & a, int & b) {
        int sv=a;
        a=b, b=sv;
    }
    const char TAB='\t';
}

void lib::rendez(int *v, int n, bool nov) {
    for (int i=0; i<n-1; i++)
        for (int j=i+1; j<n; j++)
            if (nov==(v[j]<v[i]))
                csere(v[j], v[i]);
}

void lib::kijr(const int *v, int n) {
    for (int i=0; i<n; i++)
        rendszerek::cout<<v[i]<<TAB;
    rendszerek::cout<<rendszerek::endl;
}

// -----
// main.cpp a könyvtár használatát bemutató példaprogram
#include <iostream>
#include <ctime>
#include "lib.h"
using namespace lib;

void main()
{
    int a[6];
    srand(unsigned(std::time(NULL)));
    for (int i=0; i<6; i++)
        a[i]=pow(rand(),%10,2);

    kijr(a,6);
    rendez(a,6); // növekvő sorrend
    kijr(a,6);
    rendez(a,6, false); // csökkenő sorrend
    std::cin.get();
}
```

9.5 A tárolási osztályok használata

A tárolási osztály megadásával lehetőségünk van az alapértelmezéstől eltérő élettartam és érvényességi tartomány (hatókör) kialakítására. Azok az azonosítók, amelyek tárolási osztályra **auto** vagy **regis-ter**, lokális élettartammal, míg a **static**, illetve **extern** azonosítók globális élettartammal rendelkeznek. A **typedef** szintén tárolási osztályt je-
löl a C++ nyelvben, azonban csak formai szempontból soroljuk ebbe a csoportba.

A változó- és a függvény-deklarációk forrásállományon belüli elhelyezkedése szintén hatással van a tárolási osztályra és a láthatóságra. Azok a deklarációk, amelyek min-den függvényen kívül helyezkednek el, a „külső szintű” deklarációk, míg a függvé-nyen belül megadott deklarációk – „belső szintűek”. A tárolási osztály azonosítójának pontos jelentése függ attól, hogy a deklaráció a külső vagy a belső szinten szerepel, illetve attól is, hogy változót vagy függvényt deklarálunk. Az alábbi táblázat-ban összefoglaltuk az azonosítók élettartamára és láthatóságra vonatkozó megállapí-tásokat:

Jellemzők		Élettartam		Láthatóság	
Szint	Tétel	Tárolási osztály	static	extern	extern
fájlszintű hatókör	változó-definíció	globális	korlátozva az adott állományra, a definíció helyétől a fájl végéig	globális	globális
	változó-deklaráció	extern	globális	globális	globális
	prototípus vagy függvénydefiníció	static	globális	globális	globális
blokk-szintű hatókör	változó-deklaráció	extern	globális	globális	globális
	változó-definíció	static	globális	globális	globális
	változó-definíció	extern	globális	globális	globális
Eredmény		Láthatóság			

Minden tárolási osztály esetén tisztáznunk kell a deklarált változó vagy függvény élet-tartamát és láthatóságát, illetve változók esetén az inicializálás kérdését is.

9.5.1 Az auto tárolási osztály

Azok a változók, amelyeket blokkon belül definiálunk alapértelmezés szerint automa-
tikus (**auto**) változók. Az automatikus változók a függvények belső változói, amelyek
akkor kezdenek el létezni, amikor a függvényt meghívjuk. A függvényből való kilépés
során pedig megszűnnek. (A függvény paramétereit is hasonló módon kezeli a rend-
szer.)

Mint ismert, az összetett utasítások (a blokkok) egymásba ágyazhatók. Mindegyik
blokk egyaránt tartalmazhat definíciókat (deklarációkat) és utasításokat:

```
{
    Definíciók, deklarációk és
    utasítások
}
```

Valamely automatikus azonosító ideiglenesen láthatatlanná válik, ha egy belső blokk-
ban ugyanolyan névvel másik változót definiálunk. Ez az elfedés a belső blokk vé-
géig terjed, és nincs mód arra, hogy az elfedett (létező) változóra hivatkozzunk:

```
void main()
{
    int sum = 10;
    // az int típusú sum értéke 10 *
    {
        float sum=3.1416;
        // a float típusú sum értéke 3.1416 *
    }
    // az int típusú sum értéke 10
}
```

Mivel az automatikus tárolók a blokkból kilépve megszűnnek, ezért alapvetően hibás
elgondolás olyan függvényt írni, amely egy automatikus változó címevel (vagy refe-
renciájával) tér vissza. Ha a függvényen belül kívánunk helyet foglalni olyan tároló
számára, melyet a függvényen kívül használunk, akkor a dinamikus memóriafoglalás
eszközeit kell alkalmaznunk. Az alábbi két függvény közül a *fordit1()* hibás (az el-
mondottak értelmében), míg a *fordit2()* a helyes megoldást tartalmazza. A függvények
a kapott karaktersorozatot megfordítják, és egy másik területen adják vissza:

```
// Hibás megoldás!
char * fordit1(const char * s) {
    char a[80]; // az a tömb automatikus változó!
    int i = strlen(s);
    a[i]=0;
    while (*s)
        a[--i] = *s++;
    return a;
}
```

```
// Hélyes megoldás
char * fordlt2(const char * s) {
    // az a mutató ugyan automatikus, de az általa kifejezett
    // terület dinamikus helyfoglalású!
    char * a = new (nothrow) char[strlen(s)+1];
    int i = strlen(s);
    if (ia)
        return NULL; // ha nincs hely
    a[i]=0;
    while (*s)
        a[--i] = *s++;
    return a;
}
```

Az **auto** változók inicializálása minden esetben végbemegy, amikor a vezérlés a blokkhoz kerül. Azonban csak azok a változók kapnak kezddőértéket, amelyek defini-
ciójában szerepel kezddőértékadás. (A többi változó értéke határozatlan!). Mivel az
automatikus változók esetén az inicializáló kifejezés kiértékelése futási időben törté-
nik, ezért tetszőleges kifejezés megadható (például függvényhívás is):

```
int fv(void) {
    double pi = asin(1)*2;
    int lepes = 20;
    double lrad = 2*pi/lepes;
    int a;
    int * pa = &a;
}
```

Meg kell jegyeznünk, hogy az ANSI C++ az automatikus tömb- és struktúrátípusú vál-
tozók tetszőleges kifejezéssel történő inicializálását is lehetővé teszi:

```
double b[3]={sin(12), cos(12), sqrt(12)};
```

9.5.2 Az extern tárolási osztály

A C++ program általában egy sor külső azonosítót használ. A *külső* kifejezést a függ-
vények paramétereit és automatikus változóit jellemző *be/ső* kifejezéssel ellentétes ér-
telmben használjuk. C++ nyelv külső azonosítói a függvényeken kívüli definiált vál-
tozók és függvények nevei. Azok a külső változók és függvények, amelyek definíció-
jában nem adunk meg tárolási osztályt, alapértelmezés szerint **extern** tárolási osztály-
lyal rendelkeznek. (Természetesen az **extern** közvetlenül is megadható.)

Az **extern** változók és függvények élettartama a programba való belépéstől a program
befejezésig terjed. Azonban a láthatósággal lehetnek problémák. Egy adott modulban
definiált függvény csak akkor érhető el egy másik modulból, ha abban szerepeltekük a
függvény prototípusát (például deklarációs állomány beépítésével).

Bonyolultabb a helyzet a változók esetén. Ha az egyik modulban egy változót tárolási osztály nélkül szerepeltetünk a függvényeken kívül, akkor ez a változó definícióját jelelenti (függetlenül attól, hogy adunk-e kezdőértéket vagy sem). A változó másik modulból való eléréséhez meg kell adnunk a változó deklarációját az **extern** tárolási osztályt követően. Az elmondottakat jól szemlélteti az alábbi példa:

```
// modul1.cpp
// prototípus - deklaráció
double Korkerulet(double r);
// definíció
double Korkerulet1(double r)
{
    return r*Korkerulet(r)/2;
}
// definíció
double Korkerulet2(double r)
{
    return 2*r*pi;
}
// deklaráció
extern double pi;
// definíció
double Korkerulet2(double r)
```

A szabványos C++ nyelv (a C-vel ellentétben) lehetővé teszi, hogy a statikus élettartamú változóknek futásidejű kifejezést adjunk kezdőértékként.

9.5.3 A static tárolási osztály

A **static** tárolási osztály mind külső, mind pedig belső szintű azonosítókkal együtt használható. Ha külső szintű azonosítók előtt szerepel, akkor az azonosítók láthatóságát a forrásállományra korlátozza. Ha pedig belső szintű nevek előtt adjuk meg, a nevek élettartamát automatikusról globálisra módosítja.

Amikor több modulból álló programot fejlesztünk, a modulis programozás elvének megvalósításához nem elegendő, hogy vannak közös változóink. Szükségünk lehet olyan modul szinten definiált változókra és függvényekre is, amelyek elérését a modutra korlátozhatjuk (információrejtés). Ezért **extern** és **static** tárolási azonosítókat egyaránt használunk a megfelelő modulszerezet kialakításához. A C++ szabvány a statikus külső változók helyett a névtelen névtérület használatát javasolja.

Példaként készítsünk olyan modult, amely pszeudoveletlen szám előállítására használható! Több modulból álló program esetén feltétlenül szükség van arra, hogy az egyes modulok elején információkat helyezzünk el a fájlok tartalmáról. A modulban definiált **extern** függvények deklarációját a *random.h* deklarációs állomány tartalmazza.

```
// random.h
// Pszeudovéletlen számok sorozatának előállítására
// szolgáló modul globális deklarációt.
#define randomH
namespace nsrandom
{
    void set_random(unsigned short int);
    unsigned short int random(void);
}
#endif
```

A *random.cpp* állomány két kívülről is elérhető függvényt és egy modulszintű változót tartalmaz:

```
// random.cpp
// Pszeudovéletlen számok sorozatának előállítása.
// set_random - a kívülről érték beállítása,
// random - a következő véletlen szám lekérdezése.
#include "random.h"
using namespace nsrandom;

// modulszintű változók és konstansok
// definiálása (static helyett!)
namespace {
    const short SZORZO = 97;
    const int OSZTO = 0x10000;
    const short NOVELES = 59;
    unsigned short int pseudo=0;
}

// A kívülről érték beállítása
void set_random(unsigned short int init) {
    pseudo = init;
}

// A következő véletlen szám előállítása
unsigned short int nsrandom::random(void) {
    pseudo = (SZORZO * pseudo + NOVELES) % OSZTO;
    return pseudo;
}
```

Végezetül tekintsük a *randmain.cpp* programot, amelyben kockadobás szimulációjához használjuk a fenti *random()* függvényt!

```
// randmain.cpp
// A hatlapú kocka dobásának szimulációjára
// - a példában a kockát 5-ször dobjuk
#include <iostream>
using namespace std;
#include "random.h"
using namespace nsrandom;
```



```
void main()
{
    set_random(2005); // a generátor inicializálása
    cout<<"Otszor dobunk a kockával: "<<endl;
    for (int i=0; i<5; i++)
        cout<<i<<" . dobás \t"<<random()%6+1<<endl;
    cin.get();
}
```

A **static** tárolási osztály másik, a C++ szabvány által is támogatott alkalmazási területe a statikus belső szintű változók definiálása. Az ilyen változók láthatósága a definíciót tartalmazó blokkra korlátozódik, azonban a változó a program indításától a programból való kilépésig él. A statikus változók inicializálása szintén egyetlen egyszer, a változó létrehozásakor megy végbe. Ezért ezek a statikus lokális változók a blokkból való kilépés után (a függvényhívások között) is megőrzik értéküket.

Az előző példánk véletlenszám-generátort egyetlen függvény felhasználásával is megvalósíthatjuk:

```
unsigned short int random(unsigned short int init=0)
{
    static const short SZORZO = 97;
    static const int OSZTO = 0x10000;
    static const short NOVELES = 59;
    // statikus lokális változó definiálása 0 kezdőértékkel
    static unsigned int pseudo=0;
    if (init) // Ha az init nem 0
        pseudo = init;
    else
        pseudo = (SZORZO * pseudo + NOVELES) % OSZTO;
    return pseudo;
}
```

Az *random()* függvényt kétféleképpen kell hívni. Ha 0-tól különböző argumentummal hívjuk, akkor a véletlenszám-sorozatot kezdőértékét állítjuk be. A következő véletlen szám lekérdezéséhez 0 értékű argumentummal, vagy argumentum nélkül kell az *random()* függvényt aktivizálnunk.

A statikus változók kezdőértékkel való ellátására ugyanazok a szabályok vonatkoznak, mint az **extern** változókra. A **static** változók inicializálása a programba való belépés során egyszer megy végbe. Amennyiben nem adunk meg kezdőértéket a definícióban, úgy a fordító automatikusan 0-val inicializálja a változót (feltölthetjük annak területét nullás bájtokkal). C++-ban a statikus változók kezdőértékeként szintén tetszőlegesen kifejezést használhatunk.

9.5.4 A `register` tárolási osztály

A `register` tárolási osztály csak belső szintű azonosítókhoz (automatikus lokális változókhoz és paraméterekhez) használható. A `register` kulcsszó megadásával közöljük a fordítóval, hogy az adott változót gyors eléréssel szeretnénk kezelni. Ennek érdekében a fordító (amennyiben módjában áll kérésünket teljesíteni) a processzor regiszterében hozza létre a változót. Ha nincs szabad felhasználható regiszter, akkor a tároló `auto` változóként jön létre.

A fenti megoldásból következik, hogy nincs mód annak leképezésére, hogy a tárolást végül is hol valósította meg a fordító (memóriában vagy regiszterben). A regiszterek adattárolási kapacitása processzoronként eltérő, ezért implementációnként különbözőnek azok a típusok, amelyeket regiszterben tárolhatunk. A legtöbb C++ fordító az egész jelleget, a mutató- és a referenciátípusokat tárolja regiszterben.

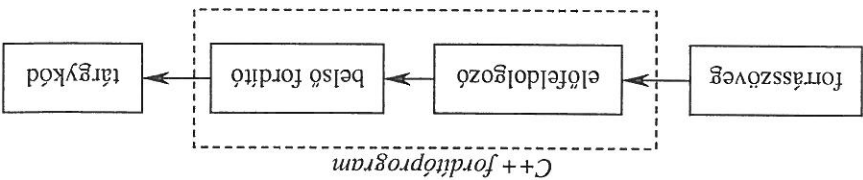
Az alábbi `cser` függvény a korábban bemutatott megoldásnál potenciálisan gyorsabb működésű. (A potenciális szót azért használjuk, mivel nem ismert, hogy a `register` előírások közül hányat teljesít a fordító.)

```
void cserereg(int &a, register int &b) {
    register int c = a;
    a = b;
    b = c;
}
```

A `register` változók élettartama, láthatósága és inicializálásának módja megegyezik az `auto` tárolási osztályú változókével.

10. Az előfeldolgozó (preprocesszor)

Minden C++ fordítóprogram szerkesztést képezi az ún. előfeldolgozó (*preprocesszor*). A fordítóprogram gyakorlatilag nem azt a forráskódot fordítja, amit mi beépeltünk, hanem az előfeldolgozó által előállított szöveget dolgozza fel.



A C++ fordítók többségében ez a két jól elkülöníthető fordítási szakasz nem válik külön, azaz nem jelenik meg az előfordító kimenete valamilyen szöveges állományban. A fentiekben bemutatott működés egyben az előfordító használatának előnyét és hátrálya is jelenti. A fejezetben ismertetésre kerülő megoldások a C++ program olvasható formában történő előállítását támogatják. A hátrány pedig abban áll, hogy a programozó általában nem látja azt az előfordított kódot, amiből a futó program előáll. Ebből következik, hogy bizonyos előfordításból származó programhibák kiderítése nem egyszerű feladat.

Nem kell tehát csodálkoznunk azon, hogy a szabványos C++ nyelv tartalmaz olyan eszközöket, amelyek segítségével az előfordító bizonyos elemeit igazi C++ program-elemekkel helyettesíthetjük. Nevezetesen, a *#define* konstansok helyett *const* konstansokat használhatunk, a *#define* makrók helyett pedig *inline* függvényabszolonokat készíthetünk.

Az előfeldolgozó olyan sororientált szövegfeldolgozó (makrónyelv), amely semmi sem „tud” a C++ nyelvről. Ez két fontos következménnyel jár:

- az előfeldolgozó által elvégzett minden művelet egyszerű szövegkezelés (figygetlenül attól, hogy a C++ nyelv kulcsszavai, kifejezései vagy változói szerepelnek benne).
- az előfeldolgozó nem csúszhatnak át másik sorba, ha csak nem jelölünk ki folytatósor), mint az egyéb C++ utasításokat (tehát egy sorba csak egy utasítás kerülhet, és a paraméterek nem csúszhatnak át másik sorba, ha csak nem jelölünk ki folytatósor),

A preprocesszornak szóló parancsokat a sor elején (esetleg szöközők és/vagy tabulátorok után) álló *#* karakter jelzi. Az előfeldolgozót leggyakrabban szöveghehelyettesítésre (*#define*) és szöveges fájl beépítésére (*#include*) használjuk. Ugyancsak jól alkalmazható a program részeitnek feltételről függő fordítására is (*#if*). A preprocesszor-parancsokat szokás direktíváknak is nevezni.

10.1 Állományok beépítése a forrásprogramba

Az *#include* direktíva utasítja az előfeldolgozót, hogy az utasításban szereplő szöveges állomány tartalmát építse be a programunkba. (A beépítés helyét a direktíva elhelyezkedése határozza meg.) Általában a deklarációkat és makrókat tartalmazó ún. fejléceállományokat (*header file*-okat) szoktuk beépíteni a programunk elején, azonban teljesülges szöveges állományra használható a művelet. A beépítési utasítás általános alakja:

```
#include <fajlnev>
//illetve
#include "fajlnev"
```

ahol a *fajlnev* a beépítendő állomány neve. Az első formát általában a C++ rendszer szabványos fejléceállományainak (*iostream*, *cmath*, *cstring*, *cstdint* stb.) beépítésére használjuk. A második pedig a saját magunk készített fájlok beépítésére szolgál.

Mindig problémát jelent annak eldöntése, hogy mi is szerepeljen a beépített állományokban. Az alábbi táblázat segíthet e kérdés megválaszolásában:

Mi lehet a fejléceállományokban? Példa

Megjegyzések	<i>/* megjegyzés */</i>
Feltételes fordítási direktívák	<i>#ifdef DEBUG</i>
Makró-definíciók	<i>#define UNIX</i>
<i>Include</i> direktívák	<i>#include <iostream></i>
Felsorolások	<i>enum Valasz {nem, igen, talan};</i>
Konstansdefiníciók	<i>const double pi=3.1415265;</i>
Nével ellátott névterületek	<i>namespace nsrandom { /* ... */ }</i>
Névedklarációk	<i>class Vektor;</i>
Tipusdefiníciók	<i>struct Complex { double re, im; };</i>
Változó-deklarációk	<i>extern int x[];</i>
Függvény-deklarációk (prototipusok)	<i>void csere(int &, int &);</i>
<i>Inline</i> függvény-definíciók	<i>inline int sqr(int a) {return a*a;}</i>
<i>Template</i> deklarációk	<i>template <class Tipus> class Vektor;</i>
<i>Template</i> definíciók	<i>template<class Tipus> class Vektor { /*...*/};</i>

Néhány olyan elemet külön is megemlítenék, amit nem szabad *include* fájlba helyezni:

- nem *inline* függvények definícióját,
- változó-definíciót,
- névtelen névter definícióját.

10.2 Feltételes fordítás

A feltételes fordítás lehetőségeinek használatával elérhető, hogy a forrásprogram bizonyos részei csak adott feltételek teljesülése esetén kerüljenek be az előfeldolgozó által előállított programba. A feltételes fordítandó programrészek kijelölésére többféle szerkezet közül választhatunk. A különböző megoldásokat az *#if*, *#ifdef*, *#ifndef*, *#elif*, *#else* és *#endif* preprocesszor-direktívákkal állíthatjuk elő. Az *#if* utasításban a feltétel megadására konstans kifejezéseket használhatunk, melyek nem nulla és 0 értéke jelöli az igaz, illetve a hamis feltételt, például:

```
#if 'z' - 'a' == 25
```

A feltételek másik csoportjával azt ellenőrizhetjük, hogy a megadott szimbólum definíált-e vagy sem. Ehhez a feltételben a *defined* operátort használjuk, amely 1 értékkel tér vissza, ha az operandusa létező szimbólum:

```
#if defined szimbólum
```

vagy

```
#if defined(szimbólum)
```

Mivel gyakran használunk ehhez hasonló vizsgálatokat, ezért valamely szimbólum létezésének tesztelésre külön előfeldolgozó utasítás áll a rendelkezésünkre:

```
#ifdef szimbólum
```

Az *#if* segítségével azonban bonyolultabb feltételek is megfogalmazhatók:

```
#if defined szimbólum && ('z' - 'a' == 25)
```

Nézzük először az egyszerűbb szerkezetet, amellyel két programrész közül választhatunk!

```
#if konstans_kifejezés
programrész1
#else
programrész2
#endif
```

Ez a szerkezet jól használható ún. hibakeresési (*debug*) információk beépítésére a programba. Ekkor egyetlen szimbolikus konstans 1 vagy 0 értékével jelölhetjük ki a lefordítandó programrészeket.

Az alábbi példában szereplő osztást végző függvény a program fejlesztési szakaszában (*#define DEBUG 1*) kírja a paraméterek értékeit, és 0-val való osztás esetén hagyja hibajelzéssel kilépni a programot. Az „éles” verzióban (*#define DEBUG 0*) azonban

valahogy kivédi a 0-val való osztást. A *DEBUG* szimbólumot a program elején látjuk el a megfelelő értékkel:

```
#define DEBUG 1

double osztas(int a, int b) {
    #if DEBUG
        cout<<"osztas - a = "<<a<<endl;
        cout<<"osztas - b = "<<b<<endl;
    #else
        if (b == 0) {
            a = MAXINT;
            b = 1;
        }
    #endif
    return (double) a/b;
}
```

A fenti szerkezet helyett jobb a *DEBUG* szimbólum definiáltságát vizsgáló megoldást használni, melyhez a *DEBUG* érték nélkül is definiálható:

```
#define DEBUG

...
#if defined(DEBUG)
    cout<<"osztas - a = "<<a<<endl;
    cout<<"osztas - b = "<<b<<endl;
#else
    if (b == 0) {
        a = MAXINT;
        b = 1;
    }
#endif
```

Az alábbi két-két vizsgálat ugyanazt az eredményt szolgáltatja:

```
#if defined(DEBUG)
...
#endif

#ifdef DEBUG
...
#endif
```

Illetve a fordított megoldás:

```
#if !defined(DEBUG)
...
#endif

#ifndef DEBUG
...
#endif
```

Bonyolultabb szerkezetek kialakításához egy másik preproceszor-utasítást ajánlott használni, ami többirányú elágaztatás megvalósítását teszi lehetővé:

Az előfeldolgozó minden programsort átvizsgál, hogy az tartalmaz-e valamilyen konstansokat, kulcsszavakat, illetve gyakran használt utasításokat és kifejezéseket. Az így definiált konstansokat reprezentáló makrókat szimbolikus konstansnak nevezzük. Altlában a kifejezéseket és utasításokat megvalósító definíciókat hívjuk makrónak (függvényyszerű makró). A makrónevekre ugyanaz a képzési szabály vonatkozik, mint más azonosítókra. Azért, hogy a preprocesszor számára definiált szimbólumok a C++ forrásnyelvi szövegben jól elkülönüljenek a programban használt azonosítóktól, a makróneveket csupa nagybetűvel ajánlott írni.

10.3 Makrók használata

Az *#include* utasítások használata során előfordul, hogy ugyanazt az állományt több-ször építjük be a programunkba, ami programhibát okozhat. Nézzük meg, hogyan old-hatjuk meg ezt a problémát! A következő megoldást minden makrókat definiáló fejállományban ajánlott alkalmazni. A deklarációs fájlt első beépítésekor létrejön egy szimbólum (például *fajlinevH*), melynek létezését vizsgálva elkerülhető az ismételt be-építés:

```
#if konstans_kifejezes1
    programész1
#elif konstans_kifejezes2
    programész2
#elif konstans_kifejezes3
    programész3
#else
    programész4
#endif
```

10.3.1 Szimbolikus konstans makrók készítése

Ebben az esetben a *#define* direktíva egyszerűbb alakját alkalmazzuk:

```
#define azonosító helyettesítő szöveg
```

Nézzünk néhány példát a szimbolikus konstansok definiálására!

```
#define EOS '\0'
```

```
#define TRUE 1
```

```
#define FALSE 0
```

```
#define NOT !
```

```
#define BOOL int
```

```
#define IGEN 1
```

```
#define NEM ! IGEN
```

```
#define URES
```

```
#define STR ""
```

```
#define UDV "üdvözlök dichó lovag ...!"
```

```
#define PMBERT 1024
```

```
#define HA !F
```

```
#define KIR cout
```

A fenti makrókat használva eléggé furcsa kinézetű (működő) C++ programot írhatunk. A kifejtés menetének érdekeltetés érdekében nézzük meg a

```
HA (NEM)
```

sor feldolgozásának lépéseit!

```
!F (NEM)
```

```
→
```

```
!F (! IGEN)
```

```
→
```

```
!F (! !)
```

10.3.2 Függvényyszerű makrók készítése

A makrók felhasználási lehetőségeit lényegesen megnövelik a paraméterezett makrók, melyek definíciójának általános formája:

```
#define azonosító(paraméterlista) helyettesítő szöveg
```

A makró hívása:

```
azonosító(argumentumlista)
```

A makróhívásban szereplő argumentumok számának meg kell egyeznie a definícióban szereplő paraméterek számával. (Lehet paraméterek nélküli makró is készíteni.) Az alábbi példaprogramban bemutatjuk néhány gyakran használt makró definícióját:


```
// x abszolút értékének meghatározása
#define abs(x) ((x) < 0 ? -(x) : (x))

// a és b maximumának kiszámítása
#define max(a,b) ((a) > (b) ? (a) : (b))

// A és B minimumának kiszámítása
#define min(A,B) ((A) < (B) ? (A) : (B))

// Négyzetre emelés
#define sqr(x) ((x) * (x))

// Két egész szám tartalmának felcserélése
// (csak egész balérték argumentummal működik helyesen)
#define csere(a,b) { int c; c = a, a=b, b=c; }
```

A makrókra jellemző, hogy általában tetszőleges típusú argumentummal meghívhatók. (A fenti *csere()* makró kivételnek számít.) A makró törzse az argumentumok aktuális szövegeztékének behelyettesítése után bemásolódik a hivatkozás helyére. Nézzük meg közelebbről az *sqr(a)* hivatkozást, melyet az előfordító az alábbi kifejezéssel helyettesít:

(a) * (a)

A makró törzsében a paramétereket általában zárójelben kell használnunk, ellenkező esetben bizonyos argumentumokkal aktivizálva hibás eredményt kapunk. Példaként nézzük meg az *sqr()* makróról úgy is, hogy ne legyenek zárójelben a paraméterek:

```
#define sqr2(x) ( x * x ) // hibás!
```

Hasonlítsuk össze az *sqr* és az *sqr2* hívását *a* és *a+1* argumentumokkal!

<i>sqr</i>	<i>sqr2</i>
$(a) * (a)$	$(a * a)$
$(a+1) * (a+1)$	$(a+1 * a+1)$
a	$a+1$

A zárójelzésnek azonban kellene legyen következményei is lehetnek, mint például amikor valamilyen léptető operátor szerepel a makró argumentumában:

```
int a=5;
cout<<sqr(a++); // 30
cout<<a; // 7
```

A programészlet lefutásakor a kiírt értékek 30 és 7. A hiba a kifejtett makró kiértékelésében keresendő, amely mellékhatásokat tartalmaz:

(a++) * (a++)

A mellékhatás miatt a kifejezés kitértékeltése implementációfüggő. Sajnos általánosan is elmondható, hogy nem szabad léptető operátort tartalmazó kifejezést makrónak átadni. A fenti programrészt kétféle módon javítható. Az egyszerűbb első megoldást akkor kapjuk, ha a léptetés műveletét külön hatjuk végre:

```
int a=5;
cout<<sqrt(a); // 25
a++;
cout<<a; // 6
```

A C++ nyelvben ajánlott másik megoldás, ha a makró helyett **inline** függvényt, illetve függvénysablonnt használunk a négyzetre emelés elvégzésére:

```
inline int square(int x) {
    return x * x;
}

template<class tip>
inline tip sqr(tip x) {
    return x * x;
}
```

Az eddigi példánkban a helyettesítést csak különálló paraméterek esetén végzi el az előfeldolgozó. Vannak esetek, amikor a paraméter valamely azonosítónak része, vagy az argumentum értékét sztringként kívánjuk felhasználni. Ha a helyettesítést ekkor is el kell végezni, akkor a ##, illetve a # makróoperátorokat kell használnunk.

A ## operátor megadásával két szintaktikai egységet (*token*) lehet összerakoztatni oly módon, hogy a makró törzsében a ## operátort helyezzük a paraméterek közé.

A alábbi példában szereplő *show()* makró tetszőleges x-szel kezdődő nevű numerikus változó értékét *double* típusúvá konvertálva jeleníti meg:

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

#define show(a) cout<<(double)(x##a)

void main()
{
    double x1 = 10;
    int xyz = 20;
    show(yz); // cout<<(double)(xyz);
    show(1); // cout<<(double)(x1);
    cin.get();
}
```

A # karaktert a makró paramétere elé helyezve, a paraméter értéke idézőjelek között (sztringként) helyettesítődik be. Ezzel a megoldással sztringben is lehetséges a be-lyettesítés, hisz a fordító az egymás mellett álló sztringliterálokat egyetlen sztring-konstanssá kapcsolja össze. Az alábbi példában a *megjelenit()* makró segítségével tetszőlegesen változó neve és értéke megjeleníthető:

```
#include <iostream>
using namespace std;
#define megjelenit(n) \
    cout<<#n" változo értéke: "<<n<<endl;

void main() {
    int a = 23;
    char *p = "C++ nyelv";
    double pi = 3.14259265;
    megjelenit(a);
    megjelenit(p);
    megjelenit(pi);
    cin.get();
}
```

a változo értéke: 23
 p változo értéke: C++ nyelv
 pi változo értéke: 3.14259

Ha makró által lefoglalt azonosítót meg szeretnénk szüntetni, akkor az *#undef* direktí-
vát kell használnunk. A makró új tartalommal történő átdefiniálása előtt mindig meg-
kell szüntetnünk a régi definíciót. Az *#undef* utasítás nem jelez hibát, ha az azonosító
nincs definiálva. Az *#undef* használata:

```
#undef azonosító
```

10.3.3 Előre definiált makrók

Az ANSI C++ szabvány az alábbi, előre definiált makrókat tartalmazza. (Mindegyik
azonosító két aláhúzással kezdődik és végződik.) Az előre definiált makrók nevét
nem lehet sem a *#define* sem pedig az *#undef* utasításokban szerepeltetni. Az előre
definiált makrók értékét beépíthetjük a program szövegébe, de feltételes fordítás
feltételeként is felhasználhatjuk.

Makró	Leírás	Példa
DATE	A fordítás dátumát tartalmazó sztringliterál.	"Dec 23 2004"
TIME	A fordítás időpontját tartalmazó sztringliterál.	"12:23:07"
FILE	A forrásfájl nevét tartalmazó sztringliterál.	"preproc.cpp"
LINE	A forrásállomány aktuális sorának sorszámát tartalmazó számkonstans (1-től sorszámoz).	23
STDC	A értéke 1, ha a fordító ANSI C++ fordítóként működik, különben nem definiált.	
cplusplus	A értéke 1, ha C++ forrásállományban tesszeliük az érté- két, különben nem definiált.	

10.4 A #line, az #error és a #pragma direktívák

Több olyan segédprogram is létezik, amely valamilyen speciális nyelven megírt programot C++ forrásprogrammá alakít át (programgenerátor). A *#line* direktíva segítségével elérhető, hogy a C++ fordítóprogram ne a C++ forrásszövegben jelezze a hiba sor-számát, hanem az eredeti speciális nyelven megírt forrásfájlban. (A *#line* utasítással beállított sorszám és állománynév a *__LINE__* illetve a *__FILE__* szimbólumok értékében is megjelennek.) A direktíva használata:

```
#line kezdősorszám
```

vagy

```
#line kezdősorszám "fájlnev"
```

Az *#error* direktívát a programba elhelyezve fordítási hibaüzenet jeleníthetünk meg, amely az utasításban megadott szöveget tartalmazza. Az utasítás általános formája:

```
#error hibaüzenet
```

Az alábbi példában a fordítás hibaüzenettel zárul, ha nem C++ módban dolgozunk:

```
#if defined(__cplusplus)
#error A fordítás csak C++ módban végezhető el!
#endif
```

A *#pragma* direktíva a fordítási folyamat implementációfüggő vezérlésére szolgál. (A direktívához semmilyen szabványos megoldás sem tartozik.) Ha a fordító valamilyen ismeretlen *#pragma* utasítást talál a programban, akkor azt figyelmen kívül hagyja. Ennek következtében a programok hordozhatóságát nem veszélyezteti ez a direktíva. Az utasítás általános alakja:

```
#pragma parancs
```

Például, a struktúrátagok 8-bájtos határta igazítása, illetve a megadott sorszámu figyelmeztető fordítási üzenetek megjelenítésének tiltása az alábbi formában végezhető el:

```
#pragma pack(8)
#pragma warning( disable : 2312 79 )
```

Létezik még az üres direktíva is (*#*), amelynek semmilyen hatása sincs a fordításra:

```
#
```

Az objektum-orientált C++ nyelv

II.

11. Bevezetés az objektum-orientált C++ nyelvbe

Az objektum-orientált programozás (OOP) a gondolkozást, cselekvést közeli programozási mód. Egy objektum-orientált programozási nyelv sokkal strukturáltabb, modultartásabb és absztraktabb, mint a hagyományos programozási nyelvek.

Ellentétben a hagyományos programozási nyelvekkel, mint például a C, nem a műveletek (funkciók) megalkotása áll a programozás központjában, hanem az egymással kapcsolatban álló programegységek (objektumok) hierarchiájának megtervezése.

Míg a hagyományos programozási nyelvek használata során az adatok csak másodlagos szerepet töltenek be a rajtuk elvégzendő műveletekkel (függvények) szemben, addig az OOP nyelvben az adatokat és adatokat elvégzendő műveleteket egyenrangúan, zárt egységben kezeljük. Ezeket az egységeket objektumoknak nevezzük. Az adatok és az adatokat kezelő függvények (*metódusok*) összeépítése (*encapsulation*) nagyon fontos sajátossága minden objektum-orientált nyelvnek. A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (*class*) nevezzük. Az objektum tehát valamilyen osztálytípussal definiált változó, amelyet más szöhasználattal az osztály példányának (*instance*) nevezünk.

Az így kialakított osztályok továbbá, a nagyméretű programok kialakításához elengedhetetlen lehetőséggel, az adatrejtés (*data hiding*) képességgel rendelkezők. Ez azt jelenti, hogy az osztály tagjainak elérhetősége szabályozható a **private** és a **public** kulcsszavak felhasználásával.

```
// a Vektor osztály deklarációja
class Vektor {
    private:
        int x, y;
    public:
        void init(int a, int b);
        int getx();
        int gety();
};

// a Vektor osztály definíciójához meg kell adnunk az összes
// függvény definícióját is! (Itt nem szerepelnek!)
void main() {
    Vektor v1; // a Vektor típusú objektumpéldány létrehozása
    // hivatkodás az objektum tagjaira
    // a public elérési mód tagfüggvény hívása - OK.
    v1.init(3, 4);
    // HIBA! Az x és y adattagok kívülről nem érhetők el.
    v1.x=4;
}
```

A C++ nyelvben definiált osztálytípus a nyelv szerkesztésévé tehető, az ún. *operator overloading* (operátor-átdefiniálás, túlterhelés) mechanizmusának felhasználásával. Ennek segítségével az általunk létrehozott típushoz definiálhatunk operátorokat, kon-verziókat, sőt akár specifikus I/O műveleteket is. Ugyancsak ezt a célt szolgálják az objektum automatikus inicializálását elvégző konstruktorok, illetve az objektum le-bontásban fontos szerepet játszó destruktorkonstruktorok alkalmazásának lehetősége. Ezen eszkö-zök felhasználásával az általunk létrehozott absztrakt adattípus (*Abstract Data Type* - osztály) a C++ nyelv természetes bővíthetéseként használható.

Az objektum-orientált nyelvek fontos sajátossága az öröklés (*inheritance*). Az öröklés azt jelenti, hogy már meglévő osztály(ok)ból kiindulva újabb osztályt épít-hetünk fel, amely örökli a felhasználott osztály(ok) adattagjait és tagfüggvényeit. A C++-ban azt az osztályt, amelyből az új osztályt származtatjuk és vagy alap (*base*) osztálynak, míg az új osztályt származtatott (*derived*) osztálynak nevezzük. A C++ 2.0-as változatától kezdődően használható a többszörös öröklődés (*multiple inheritance*) mechanizmusa is, amikor az új osztály származtatása során több alap-osztályból indulunk ki.

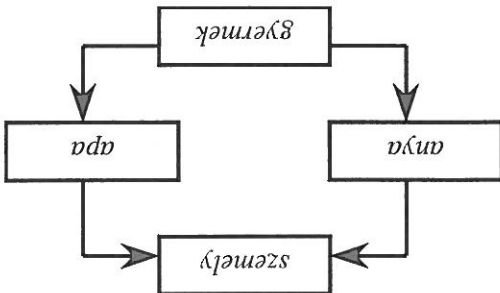
Az öröklődés során tovább szűkíthető, illetve megőrizhető az alaposztály tagjainak el-érhetősége a **private** és a **public** öröklődés felhasználásával. Az öröklődés lehetővé te-szi, hogy az egyedi objektumok helyett a problémák leírására objektumok egymásra épülő hierarchiáját használjuk.

```
// alaposztály
class személy {
};

// egyszerűsített öröklődés:
// a személy osztályból származtatott osztály
class apa : public személy {
};

// többszörös öröklődés:
// az anya és az apa osztályokból származtatott osztály
class gyermek : public anya, public apa {
};
```


A fenti példában definiált osztály-hierarchia grafikusan is ábrázolható:



Mint említettük, a származtatott osztály öröklí az alaposztály(ok) tulajdonságait (adat-tagjait és tagfüggvényeit), amelyek azonban meg is változtathatók:

- lehetőség van új tagok hozzáadására,
- a tagfüggvények újradefiniálására (*redefine*),
- az öröklött tagok elérhetőségének megváltoztatására.

A statikus osztályszerkezet a zártság tulajdonságának érvényesülése miatt nem minden esetben teszi lehetővé a tagfüggvények teljes újradefiniálását a származtatás során. Ekkor a többalakúság (*polymorphism*) mechanizmusát kell alkalmaznunk, amely virtuális (*virtual*) tagfüggvények formájában valósul meg a C++-ban. Az osztályhierarchián belül a virtuális tagfüggvények teljesen újradefiniálják az alaposztály ugyanilyen nevű, típusú és „kézjegyzű” tagjait. Ez tehát azt jelenti, hogy attól függően, hogy programunk futása során mely szintén vagyunk az objektum-hierarchiának, mindig más-más művelet (tagfüggvény) hajtódik végre. Így módon a program futása közben dől el, hogy végül is melyik tagfüggvényt kell aktivizálni. Ezt a jelenséget késői kötésnek (*late binding*) nevezzük, megkülönböztetésül a fordítás során megvalósított összesen-delesztől (*early binding*). A virtuális tagfüggvények használata alapvető követelmény minden objektum-könyvtártól.

```

#include <iostream>
using namespace std;

class alap {
public:
    void f1() { f2(); }
    void f2() { cout << "\n Alaposztály",; }
};

class uj : public alap {
public:
    void f2() { cout << "\n Származtatott osztály",; }
};
  
```

```

void main() {
    alap aobjektum;
    uj aobjektum;
    aobjektum.f1();
    aobjektum.f1();
    cin.get();
}

```

Ha a fenti példában az *f2()* tagfüggvényt nem virtuális tagfüggvényként deklaráljuk, akkor a zártság miatt az *aobjektum.f1()* és az *nobjektum.f1()* hívások hatására a program kimenete:

```

Alaposztaly
Alaposztaly

```

Amennyiben az alaposztályban az *f2()* tagfüggvényt virtuálisnak deklaráljuk,

```

class alap {
public:
    void f1() { f2(); }
    virtual void f2() { cout << "\n Alaposztaly"; }
};

```

a program kimenete a kívánt eredményt tartalmazza:

```

Alaposztaly
Szarmaztatott osztaly

```

E rövid áttekintés után részletesen megismerkedünk az objektum-orientált C++ programozás eszköztárával.

12. Osztályok

Az objektum-orientált gondolkodásmód absztrakt adattípusának (*ADT* – *abstract data type*) elkészítésére kétféle megoldás is a rendelkezésünkre áll a C++ nyelvben.

A C++ **struct** deklaráció a C struktúrátípus kiterjesztését tartalmazza, amely a C **struct** típus minden tulajdonságával rendelkezik, azonban a kiterjesztéssel alkalmazható absztrakt adattípusok definiálására. A C++ rendelkezik egy új struktúrával is, a **class** (osztály) típussal.

A **struct** és a **class** típusok adattagokból (*data member*), és ezekhez kapcsolódó műveletekből, tagfüggvényekből (*memberfunction*) épülnek fel. A **struct** és a **class** adattípusok egyaránt alkalmaznak osztályok definíciójára, azonban a tagok alapértelmezés szerinti hozzáféréseinek következtében a **class** definíció áll közelebb az objektum-orientált gondolkodásmódnak. (Könyvünkben is ezt részesítjük előnyben.) Alapértelmezés szerint a **struct** minden tagja nyilvános elérésű, míg a **class** tagjaihoz csak az osztályból lehet hozzáférni.

Az osztálydeklaráció két részből áll. Az osztály feje a **class** alapszó után az osztály nevét tartalmazza. A másik rész az osztály törzse, amelyet kapcsolós zárójelek fognak közre, és pontosvessző vagy objektumlista zár. A deklaráció az adattagokon és tagfüggvényeken kívül általában a tagokhoz való hozzáférést szabályozó **public** (nyilvános, publikált), **private** (saját, rejtett, privát) és **protected** (védezt) kulcsszavakat is tartalmaz. Az osztály törzset közfogló blokkot pontosvesszővel kell zárni.

```
class osztaly
{
    int num;
    float f;
protected:
    char *ptr;
public:
    void set (int i, float f, char *p=0) {}
    int getInt(void) {}
    float getFloat(void) {}
};
```

Az **class** és a **struct** osztályok deklarációját a C++ programban bárhol elhelyezhetjük, ahol deklaráció szerepelhet, azonban a modulszintű (fajlszintű) megadás felül megleginkább a modern programtervezési módszereknek. Ugyanezen oknál fogva csak nyilvános elérési adattagokkal rendelkező osztályok egymásba ágyazását javasoljuk, de mindig érdemes megvizsgálni, hogy az egymásba ágyazással vagy az öröklés alkalmazásával jutunk-e jobb megoldáshoz.

12.1 Adattagok

Az osztályban az adattagokat a C++ nyelv változóírhoz hasonlóan deklaráljuk, egyetlen különbség, hogy az adattagok nem tartalmazhatnak inicializációs listát. A C++ nyelv mutatók és hivatkozások esetén megengedi az ún. előrevetett osztálydeklaráció használatát. (Az éppen deklarált osztály szintén előrevetett deklarációnak számít.)

```
class osztaly1; // előrevetett deklaráció
```

```
class osztaly2
{
    osztaly1 *p01;
    osztaly2 *p02;
};
```

Amenyiben az osztályon belül egy más osztályt objektumot kívánunk adattagként elhelyezni, előzőleg szerepeltetni kell a másik osztály teljes deklarációját:

```
class osztaly1 // teljes deklaráció
{
    ...
};
```

```
class osztaly2
{
    osztaly1 o1;
    ...
};
```

12.2 Tagfüggvények

Az adattagokon műveleteket végző tagfüggvényeket az osztály törzsén belül deklaráljuk. Ezek a deklarációk a függvény prototípusát vagy pedig a teljes, implicit **inline** deklarációját tartalmazza. (Az implicit szó azt jelenti, hogy a beépített függvénydefiníció előtt nem használjuk az **inline** kulcsszót.)

```
class osztaly
{
    int num;
    float f;
protected:
    char *ptr;
public:
    void set (int i, float f, char *p=0); // prototípus
    int getint(void) { return num; } // inline definíció
    float getfloat(void) { return f; } // inline definíció
};
```

Az osztály deklarációját általában külön deklarációs állományban helyezzzük el, hogy elkerüljük a nem **inline** tagfüggvényeknek többszöri definícióját (*Osztaly.h*). A nagyobb tagfüggvényeket az osztálydefinícion kívül adjuk meg, a hatókör (::) operátor felhasználásával:

```
#include "Osztaly.h"
void Osztaly::set(int i, float f, char *p) {
    num = i;
    chis->f = f;
    ptr = p;
}
```

Az ilyen tagfüggvények szokásos helye az osztály deklarációját tároló fejállomány nevével meggyező névű C++ forrásfájl (*Osztaly.cpp*).

Megjegyezzük, hogy lokális névtérületen (függvényen belül) definiált osztály csak implicit **inline** tagfüggvényeket tartalmazhat. Az **inline** kulcsszó explicit megadásával az osztálydefinícion kívül megadott tagfüggvényeket is beépítetté tehetjük, így ezeket az osztály deklarációjával együtt a fejállományában kell elhelyeznünk.

A tagfüggvények több szempontból is különböznek a hagyományos C++ függvényektől:

- Valamely osztály tagfüggvényei mindig elérik a saját osztály (**public**, **private**, **protected**) adatait. Nem tagfüggvény függvények csak a **public** adattagokhoz férhetnek hozzá (ha nem **friend** függvényei az osztálynak.)
- A tagfüggvények az osztály érvényességi körében definiáltak, ellentétben a közönséges függvényekkel, amelyek fájlszintű vagy névtérszintű hatókörrel rendelkeznek.
- Tagfüggvényt csak ugyanazon osztálybeli tagfüggvénnyel lehet átdefiniálni (ülterhelni), hiszen az átdefiniálás (*overloading*) mechanizmusa csak azonos érvényességi körrel rendelkező függvények között használható.

Fontos megértenünk az osztály adattagjainak és tagfüggvényeinek tárolását. Ha valamilyen osztálytípussal objektumokat (osztály típusú változókat) hozunk létre:

Osztaly t1, t2;

akkor minden objektum saját adattagokkal rendelkezik (amennyiben az adattag nem statikus), a tagfüggvények egyetlen példányát pedig megosztva használják.

```
t1.set(1, 1.2);
t2.set(2, 2.3, " Lafenita ");
```

A példában feltételezzük, hogy a „*Lafenita*” karaktertetsorozat a 0x231279 címen helyezkedik el a memóriában.

<i>num</i>	1
<i>f</i>	1.2
<i>ptr</i>	0x0

t1

<i>num</i>	2
<i>f</i>	2.3
<i>ptr</i>	0x231279

t2

Felvetődik a kérdés, honnan tudja a *ser()* függvény, hogy adott esetben mely adaterről kell dolgoznia? Erre a kérdésre a fordító nem látható tevékenysége adja meg a választ. Minden tagfüggvény, még a paraméter nélküliek is, rendelkeznek egy nem látható paraméterrel (*this*), amelyben a hívás során az aktuális objektumra mutató pointer adódik át a függvénynek. A fentiekben kívül minden adattag-hivatkozás automatikusan az alábbi formában kerül be a kódba:

this->adattag

A példában szereplő osztály tagfüggvényeinek *valódi* (csak a fordító által használt) definíciója:

```
void set (int i, float f, char *p=0, Osztaly * this) {
    this->num = i;
    this->f = f;
    this->ptr = p;
}
int getInt(Osztaly * this) { return this->num; }
float getFloat(Osztaly * this) { return this->f; }
```

A *this* (ez) mutatót mi is felhasználhatjuk a tagfüggvényekben belül. Az *Osztaly::set()* függvény definíciójában *this->f=f;* azért volt rá szükség, mivel az adattag és a tagfüggvény paramétere ugyanazt a nevet (f) viseli, és a fordító először mindig a „legközelebbi” (az aktuális hatókörben fellelhető) nevet használja a fordítás során.

12.2.1 Konstans tagfüggvények és a mutable típusminősítő

A közönséges C++ függvények készítése során konstans paramétereket használunk annak érdekében, hogy a függvényen belül ne lehessen módosítani az értékeiket. Egy osztály tagfüggvényei (paraméterek helyett) az osztály adattagjain fejtik ki hatásukat. Amennyiben a konstans paramétereknek megfelelő működéshez szeretnénk jutni, konstans tagfüggvényeket kell használnunk. Konstans tagfüggvények esetében a függvényfejet a *const* kulcsszó zárja.

Konstans tagfüggvényből egyetlen adattag sem módosítható. Bizonyos esetekben előfordul, hogy az adattagok többségét nem kívánjuk megváltoztatni, azonban néhány adattag értékét módosítani szükséges. Hogy az ilyen esetekben se kelljen globális vál-

tozókát használunk, a szabvány bevezette a **mutable** típusminősítőt. A **mutable** (változókonny) minősítésű adattagokat a konstans tagfüggvényekből is megváltoztathatjuk. Az *Osztály* példánkat az elmondottak alapján módosítottuk, bevezetve egy számláló adattagot, melynek értékét a *getint()* minden hívásakor eggyel növeljük:

```
class Osztaly {
    int num;
    float f;
    mutable int szamlo; // megváltoztatható
protected:
    char *ptr;
public:
    // alapelemezett konstruktor
    Osztaly() { f=num=szamlo=0; ptr=NULL; }
    void set (int i, float f, char *p=0);
    // konstans tagfüggvények
    int getint(void) const { szamlo++; return num; }
    float getfloat(void) const { return f; }
};
```

A **mutable** kulcsszó segítségével jelölt osztálytagok minden esetben megváltoztathatók, még akkor is, ha konstansként definiáljuk az osztály példányát. Felhívjuk a figyelmet, hogy a **mutable** előírás nem alkalmazható **static** és **const** adattagok esetén.)

```
class szemely {
    const char * nev;
    mutable int eletkor;
    unsigned long lbszam;
public:
    friend void szulnap(const szemely &); // barátfüggvény
    szemely(const char *, int, unsigned long); // konstruktor
};

szemely::szemely(const char * nev, int kor, unsigned long szam) {
    this->nev = nev;
    eletkor = kor;
    lbszam = szam;
}

void szulnap(const szemely & valaki) {
    ++valaki.eletkor; // rendben
    // ez azonban hibás: ++valaki.lbszam;
}

void main() {
    const szemely Nata("Lafentita", 23, 427364899UL);
    szulnap(Nata); // rendben
    cin.get();
}
```

12.3 Az osztály tagjainak elérése

A C++ nyelv az osztály koncepcióban megvalósítja az információrejtés elvét. A **public** (nyilvános), **private** (saját) és a **protected** (védtett) alapszavakkal az egyes tagok elérését szabályozhatjuk.

```
class cls {
    // Itt helyezkednek el az alapértelmezés szerinti private
    // elérési tagok.
    // Természetesen a private: is használható a tagok előtt
protected:
    // protected adattagok és tagfüggvények
public:
    // korlátozás nélkül elérhető adattagok és tagfüggvények
};
```

Az osztály deklarációjában több **public**, **private** és **protected** rész is szerepelhet. Az egyes szekciókban az adattagok és a tagfüggvények deklarációját tetszőleges sorrendben megadhatjuk. A C++ szabványban az egyes részeket **public**, **protected** és **private** sorrendben ajánlják elhelyezni:

```
class osztaly {
public:
    void set (int i, float f, char *p=0); // prototípus
    int getint(void) { return num; }
    float getfloat(void) { return f; } // inline definíció
protected:
    char *ptr;
private:
    int num;
    float f;
};
```

Nézzük meg az elérést szabályzó egyes előírások jelentését!

- A **public** tag bárhol elérhető a programon belül, ahonnan maga az objektum elérhető. Az adatrejtés elvének érvényesüléséhez ajánlott, hogy nyilvános eléréssel csak a tagfüggvényeket deklaráljuk.
- A **protected** tagok külső függvények számára **private**, de a származtatott osztályok tagfüggvényei számára nyilvános elérésűek. Az osztálytagok védtett elérése az osztály-hierarchia kialakításánál, vagyis az öröklésnél (*inheritance*) játszik szerepet.
- A **private** tagokat csak az osztály saját tagfüggvényeiből, illetve az osztály „barátaiból” (*friend*) érhetjük el. A külső barátfüggvények és a származtatott osztály tagfüggvényei nem rendelkeznek hozzáférési joggal a **private** osztálytagokhoz (bár azok is öröklődnek). Általában az osztály adatait **private** vagy **protected** eléréssel deklaráljuk. Mindkét elérési direktíva védi a tagokat

az osztály *felhasználójától*, aki példányosítja az osztályt. A védett tagok azonban elérhetőek maradnak az osztály *további fejlesztője* számára, aki saját osztályt származtat belőle.

Mivel az adattagok általában nem elérhetők az osztály felhasználói számára, ún. nyilvános elérési tagfüggvényeket szokás definiálni az adattagokhoz (*setxxx()*, *getxxx()*). Az elérési tagfüggvények hívásával az adattagokhoz való hozzáférés, ellenében a közvetlen hozzáféréssel, az ellenőrzésünk mellett megy végbe.

12.3.1 Az osztályok friend mechanizmusa

Vannak esetek, amikor az adatrejtsési szabályok nem kívánt korlátozást jelentenek a programozó számára. A **friend** (barát) mechanizmus lehetővé teszi, hogy egy osztály **private** és **protected** tagjait nem az osztály tagfüggvényekből is elérjük. A **friend** deklarációt az osztály deklarációján belül kell elhelyeznünk. A „barát” lehet egy külső függvény, egy másik osztály adott tagfüggvénye, de akár egy egész osztály is (vagyis annak minden tagfüggvénye). Ennek megfelelően a **friend** deklarációban a függvények prototípusát, illetve az osztály nevét szerepeltetjük. Felhívjuk a figyelmet arra, hogy barátosztály esetén a „barátí viszony” nem kölcsönös, csak a deklarációban szereplő osztály tagfüggvényei kapnak korlátlan elérést a deklarációt tartalmazó osztály tagjaihoz.

```
class Sclass;

class Tclass {
    int szamlat(int x) {}
};

class Fclass {
    // A külső sum() függvény barát.
    friend long sum(void);

    // A Tclass osztálynak csak a szamlat() tagfüggvénye barát.
    friend int Tclass::szamlat(int x);

    // Az Sclass osztály barát, így annak minden tagfüggvényére
    // vonatkozik a baráti viszony. Ez a kapcsolat azonban nem
    // kölcsönös!
    friend Sclass;
};
```

Példaként tekintsük a síkbeli pontok leírásához használható egyszerűsített *Pont* osztályt! A pontok távoliságát számító művelet nem kapcsolható egyik ponthoz sem, így teljesen jogos az igény, hogy a távoliság meghatározására külső függvényt használjunk, amely paraméterként kapja a két pontot. Az adattagok gyors eléréséhez azonban szükséges a közvetlen hozzáférés biztosítása, ami a „barát” mechanizmus révén meg is valósítható.

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

```
class Pont{
friend double tavolsag(const Pont &, const Pont &);
private:
    int x,y;
public:
    void beallit(int _x, int _y) {x=_x; y=_y;}
};
```

```
double tavolsag(const Pont & p1, const Pont & p2) {
    return sqrt(pow(p1.x-p2.x,2.0)+pow(p1.y-p2.y,2.0));
}
```

```
int main() {
    Pont p,q;
    p.beallit(1,2);
    q.beallit(4,6);
    cout<<tavolsag(p,q)<<endl;
    cin.get();
}
```

12.4 Az osztály objektumai

C++ nyelvben az objektumok az osztály példányait (*instance*) jelölik, melyeket a szokásos változó-definícióval hozunk létre. Az *Osztaly* osztály példányait az alábbiak szerint definiálhatjuk:

Osztaly x; **Statikusan** létrehozott *Osztaly* típusú objektum.

Osztaly & xr=x; Referencia az x objektumhoz.

Osztaly *xp; **Mutató** *Osztaly* típusú objektumra.

xp= **new** *Osztaly*; Az *Osztaly* típusú objektum **dinamikus** létrehozása.

Az objektumtagok eléréséhez statikus esetben a pont (.) operátort, míg a dinamikusan létrehozott objektumok tagjaihoz a nyíl (->) műveletet használjuk:

x.set(1, 2.3, "C++");

xr.set(1, 2.3, "C++");

xp->set(1, 2.3, "C++");

A dinamikusan létrehozott objektumok megszüntetéséről magunknak kell gondoskodni a **delete** operátor felhasználásával:

delete xp;

A függvényargumentumként használt, illetve függvényértékként definiált objektum értéket szerint adódik át. Ezekben az esetekben a fordító ideiglenes példányt készít az objektumról, ahova az adattagokat átmásolja. Az objektum adattagjainak másolása egyszerű értékadások is megtörténik. (Az értékadás az egyetlen művelet, melyet az objektumokra minden további nélkül alkalmazhatunk.)

```
osztaly t1, t2;
t1.set(1, 2.3, "C++");
t2 = t1;
```

Felhívjuk a figyelmet arra, hogy a példában az értékadás után a *ptr* mutató mindkét objektumban ugyanazt a címet tartalmazza, mindkettő ugyanarra a ("C++") karaktertárolóra mutat. Vannak esetek, amikor ez működés nem elfogadható, ezért a C++ nyelv lehetőséget biztosít arra, hogy az alapértelmezés szerinti másolási műveletet (*X::operator=(const X&)*) átdefiniáljuk (túlterheljük). Hasonló a probléma akkor is, ha egy új objektumot már meglévővel inicializálunk:

```
osztaly a;
osztaly b = a;
```

Ekkor a meglévő objektum adattagjai szintén a fenti mechanizmus szerint másolódnak át az új objektum adattagjaiba. A másolást az ún. másoló (*copy*) konstruktor (*X::X(const&X)*) végzi, amelyet szintén átdefiniálhatunk.

Az elmondottak összefoglalásaként tekintsük az alábbi programrészletet! (Az osztály nevével (*RosszOsztaly*) azt jeleztük, hogy az adattag nyilvános elérési.)

```
class RosszOsztaly {
public:
    int a;
    void init(int x) { a = x; }

    void main() {
        RosszOsztaly x; // alapértelmezett konstruktor hívása
        RosszOsztaly y=x; // másoló konstruktor hívása
        RosszOsztaly *px;
        px = new RosszOsztaly;
        x.a = 12;
        px->a=23;
        x.init(12);
        px->init(23);
        *px = x;
        delete px;
    }
}
```

Az előzőekben már szoltunk a **this** mutató szerepéről. Most csak egy gondolat erejéig térünk vissza hozzá. Gyakran előfordul, hogy valamely tagfüggvény objektum-referenciával, objektumértékekkel vagy objektummutatóval tér vissza. Ezekben az esetekben a megfelelő függvényérték kialakításához a **return** utasításban a **this** mutatót használ-

juk. A **return*****this**; magát az objektumot (referencia esetén), vagy egy másolatot az objektumról (érték esetén) ad vissza függvényértékként. A **return this**; esetén pedig pedig az objektumra mutató pointerrel tér vissza a függvény.

12.5 Statikus osztálytagok használata

Érdekes felhasználási lehetőséget kínál a statikus adattagok alkalmazása az osztályok-ban. A **static** adattagot (a tagfüggvényekhez hasonlóan) megoszthatva használják az osztály objektumai. Ezzel elkerülhetjük azokat a problémákat, amelyek a nem osztálytag globális változók bevezetéséből származhatnak. A statikus adattag közvetlenül az osztályhoz tartozik, így az akkor is elérhető, ha egyetlen objektuma sem létezik az adott osztálynak. A statikus adattag inicializálását az osztályon kívül kell elvégezni (függétlenül az adattag elérhetőségétől).

Ha a statikus adattag **public** elérésű, akkor a programban bárholnan felhasználhatjuk az az osztály neve és a hatókör (::) operátor magadáásával. Ellenkező esetben csak az osztály példányai érik el ezeket a tagokat.

Az alábbi példában a statikus tagok használatának bemutatásán túlmenően, a konsz-tansok osztályban való elhelyezésének megoldásait (**static const** és **enum**) is szemléltetjük. Az általunk definiált matematikai osztály (*Math*) lehetővé teszi, hogy a *Sin()* és a *Cos()* tagfüggvényeket radian vagy fok mértékegységű adatokkal hívjuk:

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

class Math {
private:
    static double dFokRad;
    static bool eRadian;
public:
    typedef double valos;
    enum Egyseg { fok, rad };
    static const double Pi;
    static double Sin(double x)
    {return sin(eRadian==rad ? x : dFokRad*x);}
    static double Cos(double x)
    {return cos(eRadian==rad ? x : dFokRad*x);}
    static void Merlek(Egyseg e=rad) { eRadian=e; }
    void kilir() {cout<<Pi<<endl;}
};

// A statikus adattagok létrehozása, és a kezdőértékek beállítása
const double Math::Pi=M_PI;
double Math::dFokRad = Math::Pi/180;
bool Math::eRadian = Math::rad;
```

A példában látható módon, az osztályon belül **typedef** tárolási osztállyal definiált tállyhatókörrrel rendelkezőnek, függetlenül az **enum** kulcsszó után megadott típusnévtől (*Math::Egyseg*).

```
Math::valos
Math:::rad
Math:::fok
```

Az osztály lehetséges alkalmazását az alábbiakban láthatjuk:

```
void main () {
    Math::valos y = Math:::sin(Math:::Pi/6) ; // radiánban számol
    Math:::Mertek(Math:::fok) ; // fokokban számol
    y = Math:::sin(30) ;
    Math:::Mertek(Math:::rad) ; // radiánban számol
    y = Math:::sin(Math:::Pi/6) ;
    Math m;
    // osztálypéldány
    m.Mertek(Math:::fok) ; // m.Mertek(m.fok) ;
    y = m.Sin(30) ;
    m.Mertek(m.rad) ;
    y = m.Sin(Math:::Pi/6) ;
}
```

A **static** adattagok kezelésére általában statikus tagfüggvényeket használunk. A statikus tagfüggvényekből azonban a normál adattagokhoz nem férhetünk hozzá, mivel a paraméterek között nem szerepel a **this** mutató. A nem statikus tagfüggvényekből természetesen az osztály statikus tagjait korlátozás nélkül elérhetjük.

12.6 Osztálytagokra mutató pointerek

C++-ban egy függvényre mutató pointer akkor sem veheti fel valamely tagfüggvény címet, ha különben a típusuk és a paraméterlistájuk teljesen megegyezik. Ennek oka, hogy a (nem statikus) tagfüggvények az osztály példányain fejtik ki hatásukat. Ugyanez igaz az adattagokhoz rendelt mutatókra is. További fontos eltérés a hívás számos mutatókhoz képest, hogy a virtuális tagfüggvények mutaton keresztül hívása esetén is érvényesül a polimorfizmus. A mutatók helyes definiálásához az osztály nevét és a hatókör operátort is alkalmaznunk kell:

```
class Posztaly;
előrevetett osztálydeklaráció;

int Posztaly::*x;
x mutató egy int típusú adattagra,

int (Posztaly::*fx) (void) ;
fx mutató egy olyan tagfüggvényre mutathat,
amelyet argumentum nélkül hívunk, és int
értéket ad vissza.
```

A következő példában bemutattuk az osztálytagokra mutató pointerek használatát, melynek során mindegyik osztálytagot mutató segítségével értjük el. Az ilyen mutatók alkalmazása esetén a tagokra a szokásos operátorok helyett a **(pont csillag)*, illetve a *->*(nyíl csillag)* operátorokkal kell hivatkozni. Az adattagok és tagfüggvények címének leképezéséhez egyaránt a címre (&) operátort kell használnunk.

```
class Posztaly {
public:
    int a;
    void set(int x) { a = x; }
    int get() const { return a; }
};

void main()
{
    int posztaly:*ip;
    void (posztaly::*sp)(int);
    int (posztaly::*gp) (void) const;
    ip = &posztaly::a;
    sp = &posztaly::set;
    gp = &posztaly::get;
    posztaly o1,*o2;
    (o1.*sp) (12);
    (o1.*ip)++;
    o2 = new Posztaly;
    (o2->*ip) *= 23;
    int iv = (o2->*gp) ();
    delete o2;
}
```

```
// a mutatók beállítása
// az o1.set() hívása
// az o1.a léptetése
// az o2->a szorzása 23-mal
// az o2->get() hívása
```

13. Konstruktorok és a destruktork

A programozás során egyik legnehezebben felfedezhető hiba, ha az automatikus változók értékét kezdőértékkadás nélkül használjuk. A hiba különlegessége abban áll, hogy az ilyen változók értékét a memóriaterületen található „szemét” adja, ami sok esetben véletlenszerű. C++ objektumok esetén az inicializálásról, már az osztály készítése során gondoskodhatunk.

Ebben a részben két speciális tagfüggvényről ismertetünk meg, melyek feladata az osztály példányainak inicializálása (konstruktor), illetve a megszüntetés előtti „takartás” (destruktor).

13.1 Konstruktorok

Mint láttuk, a programban használt változók és objektumok megfelelő inicializálása fontos feladat, hisz e lépés nélkül a program viselkedése véletlenszerűvé válhat. A C nyelvben megszokott struktúra-inicializálás a **class** típusú objektum esetén csak akkor használható, ha az kizárólag **public** elérési adattagokkal rendelkezik. A C++ programokban az objektumok inicializálásának feladatát speciális tagfüggvényekkel a konstruktorokkal oldjuk meg.

A konstruktor olyan tagfüggvény, amelynek neve megegyezik az osztály nevével, és nem rendelkezik típusal. Az osztály konstruktorát a fordító minden olyan esetben automatikusan meghívja, amikor az adott osztály objektuma létrejön. A konstruktor nem rendelkezik visszatérési értékkel, de különben ugyanúgy viselkedik, mint bármely más tagfüggvény. A konstruktor áttekintésével (túllerheléssel) többféleképpen is inicializálhatjuk az objektumokat.

Fontos megértenünk, hogy a konstruktor nem foglal tártterületet a létrejövő objektum számára – ezt a fordító végzi az alapítusoknál használt módszerrel. (Ha az objektum dinamikus, akkor a **new** operátor foglal memóriaterületet.) A konstruktor feladata már lefoglalt memóriaterület inicializálása. Ha az objektum valamilyen mutató tartalmaz, ami elég gyakori megoldás, akkor természetesen a konstruktorból kell gondoskodnunk a mutató által kijelölt terület lefoglalásáról.

Az alábbi egydimenziós, egész tömböt megvalósító *Vektor* osztályban több konstruktor is definiáltunk:

Egy osztály alapértelmezés szerint két konstruktorral rendelkezik: a paraméter nélküli (*default* - $X::X()$) és a másoló (*copy* - $X::X(const \&X)$) konstruktorral. Ha valamilyen saját konstruktor készítettünk, akkor a paraméter nélküli alapértelmezett (*default*) konstruktor többé nem lesz elérhető, így azt is definiálnunk kell. Saját másoló konstruktor általában akkor készítettünk, ha valamilyen dinamikus tárterület tartozik az osztály példányaihoz (mint a példánkban).

Ha a fenti osztálydefiniációt a *vektor.h* fejláblomában tartoljuk, akkor az alábbi *vektor.cpp* fájlt tartalmazhatja a függvények definiációját:

```
#include "vektor.h"

// A paraméter nélküli alapértelmezett konstruktor 10-elemű
// tömböt hoz létre:
Vektor::Vektor(void) {
    p = new int[meret = 10];
}

// Adott méretű vektor inicializálása
Vektor::Vektor(int n) {
    p = new int[meret=n];
}

// Inicializálás másik vektorral - másoló konstruktor
Vektor::Vektor(const Vektor& v) {
    p = new int[meret=v.meret];
    for (int i = 0; i < meret; ++i) // az elemek átmásolása
        p[i] = v.p[i];
}

// Inicializálás hagyományos n-elemű vektorral
Vektor::Vektor(const int a[], int n) {
    p = new int[meret=n];
    for (int i = 0; i < meret; ++i)
        p[i] = a[i];
}

#endif
```


Az első két tagfüggvényt össze is vonhatjuk az alapértelmezett függvényargumentum kijelöléssel:

```
class Vektor {
public:
    Vektor(int n=10);
    Vektor(const Vektor& v);
    Vektor(const int a[], int n);
private:
    int *p;
    int meret;
};

Vektor::Vektor(int n) {
    p = new int[meret=n];
}
```

Az eredeti megoldásnál maradván, mind a négy konstruktor helyet foglal a memóriából a vektor elemei számára. Nézzünk néhány példát a *Vektor* típusú objektumok definiálására!

Az első (*default*) konstruktor hívódik meg az alábbi objektumok létrehozását követően. Felhívjuk a figyelmeztet arra, hogy objektumtömb (3. és 4. definíció) esetén minden tömbelem konstruktora lefut.

```
Vektor a, b;
Vektor *c = new Vektor; // c pointer egy 10-elemű vektorra
Vektor d[5]; // 5 darab 10-elemű vektor tömbje
Vektor *e = new Vektor[5]; // 5 darab 10-elemű vektor tömbje
delete c;
delete []e;
```

A második konstruktor aktivizálódik az alábbi definíciókban, melynek során 20-elemű egész tömbök jönnek létre. Az utolsó példában 3-elemű objektumtömböt hozunk létre, melynek elemei 3-, 7- és 9-elemű egész tömböket tartalmaznak.

```
Vektor f = Vektor(20);
Vektor g(20);
Vektor h = 20;
Vektor *i = new Vektor(20);
delete i;
Vektor j[]={3,7,9};
```

A következő definíciókban a harmadik (másoló) konstruktor használatára láthatunk példákat:

```
Vektor k;
Vektor l=k;
Vektor m(k);
Vektor n[3]={k, l, m};
Vektor *o=new Vektor(k);
delete o;
```

// a default konstruktor hívódik meg

Az alábbiakban a negyedik konstruktor hívódik meg:

```
int z[5] = {4, 13, 7, 26, 30};
Vektor p(z, 5);
Vektor *q=new Vektor(z, 5);
delete q;
```

A dinamikus objektumtömbök létrehozása során csak a paraméter nélküli (*default*) konstruktor hívására van mód. Ezzel szemben a statikus objektumtömbök definíciójában szereplő inicializációs lista más konstruktorok aktivizálását is lehetővé teszi.

A konstruktorok egyaránt lehetnek **public**, **private** és **protected** elérésűek. A csak **private** konstruktorokat tartalmazó osztályt rejtett osztálynak nevezzük. Az ilyen osztály objektumait csak *barát* függvényben, illetve *barát* osztályban hozhatjuk létre. A csak **protected** konstruktorokat tartalmazó osztály, amely nem rendelkezik *barátokkal* (*friends*), jól használható absztrakt alaposztályként, amelyből más osztályokat származtatunk.

Erdemes összefoglalva áttekinteni a konstruktorok használatával kapcsolatos szabályokat!

- Neve megegyezik az őt tartalmazó osztály nevével.
- Nincs visszatérési értéke (**void** típusú sem).
- Deklarációjában nem használhatjuk a **virtual**, **static**, **const**, **mutable** és **volatile** kulcsszavakat.
- A neve átdefiniálható (tültérhelhető).
- Nem öröklődik.
- Nem készíthető mutató a konstruktorhoz.
- Az objektum példány létrehozása után nem hívható.
- Az argumentum típusa és az osztálytípus közötti konverzió letiltható az **explicit** kulcsszó megadásával.

13.1.1 A konstruktorok explicit paraméterezése

Általában az egyparaméteres konstruktorral rendelkező osztályok példányainak olyan kifejezést adunk kezdőértékül, amely típusával illeszkedik a konstruktor paraméter-típusához. Ha a típusegyezés nem 100-százalékos, akkor a fordító implicit konverziókat használ a megfelelő konstruktor kiválasztásához. Az **explicit** kulcsszó megadásaival megíthatjuk az alapértelmezett implicit konverziók alkalmazását a konstrukciósorán.

```
class Vektor {
public:
    explicit Vektor();
    explicit Vektor(int n);
    explicit Vektor(const Vektor& v);
    explicit Vektor(const int a[], int n);
private:
    int *p,
    int meret;
};
```

Az előző *Vektor* osztály módosítása után az alábbi két definícióban hibát jelez a fordító, mindkettőben az *int->Vektor* átalakítást kifogásolja:

```
Vektor h = 20;
Vektor j[] = {3,7,9};
```

13.1.2 Tagobjektumok inicializálása

Ha egy objektum tagként valamely példányát tartalmazza, akkor a tag-osztály konstruktorának hívását a külső osztály konstruktorában oldjuk meg. Ez a mechanizmus a taginicializációs lista használatát jelenti, amelyet a konstruktor után két-tösponttal elválasztva adunk meg. A lista vesszővel elválasztott elemei az osztály tagjai, melyek után zárójelben áll az inicializációs argumentum.

Az alábbi példában a *Kulso* osztályon belüli *Belso* típusú objektumot inicializáljuk a *Kulso* konstruktorában.

```
class Belso {
    int cnt;
public:
    Belso(int v) : cnt(v) {} // inline konstruktor
};

class Kulso {
    int num;
    Belso obj;
public:
    Kulso(int, int); // a konstruktor prototípusa
};
```

```
Kulso::Kulso(int k, int b) : obj(b)
{
    num = k;
}
```

A taginitializációs listát csak a konstruktor-definícióban adhatjuk meg. Ezt a módszert azonban tetszőleges típusú adattag esetén is használhatjuk. A fenti konstruktor egy lehetséges másik alakja:

```
Kulso::Kulso(int k, int b) : obj(b), num(k)
{
}
```

A taginitializációs lista használata kötelező, ha az osztály referencia típusú adattagot vagy paraméterezett konstruktorral rendelkező objektumot tartalmaz:

```
class Kulso {
    int num;
    Belso & obj;
public:
    Kulso(int a, Belso & br) : obj(br), num(a) {}
};
```

Az initializációs lista feldolgozása a konstruktor törzsének végrehajtása előtt megy végbe.

13.1.3 Konstans tagok initializálása

Az osztályban elhelyezett konstans (**const**) adattagok értékének beállítására is a taginitializációs listát kell használnunk az alábbi példában bemutatott (aláhúzással kiemelt) módon:

```
class Kor {
protected:
    double r;
public:
    const double pi;
    Kor(double r=0) : pi(3.14159265), r(r) {}
    double Terulet() const { return r*r*pi; }
    double Kerulet() const { return 2*r*pi; }
};

void main() {
    Kor k1(10), k2(12);
    cout<<k1.Terulet()<<"\t"<<k1.Kerulet()<<endl;
    cout<<k2.Terulet()<<"\t"<<k2.Kerulet()<<endl;
    cin.get();
}
```

13.2 Destruktor

A fenti *Vektor* példában a konstruktorok memóriát foglalnak a vektor elemei számára, a *new* operátor felhasználásával. Ez a memória mindaddig lefoglalt marad, amíg fel nem szabadítjuk. Erre a célra a C++ biztosít egy speciális tagfüggvényt, a destruktort, amelyben gondoskodhatunk a lefoglalt tártérület felszabadításáról. A destruktort nevével hullám karakterrel (*ilde*, ~) egybeépített osztálynévként kell megadni. A destruktort, akárcsak a konstruktor, szintén nem rendelkezik visszatérési típussal.

A destruktort tartalmazó *Vektor* osztályban a destruktort felszabadítja a konstruktor által lefoglalt memóriát, amikor az objektum megszűnik:

```
class Vektor {
public:
    Vektor();
    Vektor(int n);
    Vektor(const Vektor& v);
    Vektor(const int a[], int n);
private:
    ~Vektor() { delete[] p; } // inline destruktort
    int *p;
    int meret;
};
```

A destruktóban nem vizsgáljuk, hogy a *p* mutató rendelkezik-e nullától (*NULL*) különböző értékkel:

```
~Vektor() { if (ip) delete[] p; }
```

Ennek oka, hogy a **delete** műveletek *NULL* értékű mutatók esetén is helyesen működnek, és ezt jóval gyorsabban teszik, mint az **if** utasítás. A programfejlesztés biztonságosabbá tehető a mutatók létrehozás utáni „lenullázásával”.

Ha az osztály rendelkezik destruktortal, a fordító minden olyan esetben meghívja azt, amikor az objektum érvényessége megszűnik. Kivételt képeznek a **new** operátorral dinamikusan létrehozott objektumok, melyek destruktort csak a **delete** operátor megadásával aktivizálhatjuk. Szintén fontos megjegyeznünk, hogy a destruktort nem magát az objektumot szünteti meg, hanem automatikusan elvégez néhány általunk megadott „takarítási” műveletet. (A destruktort közvetlenül is meghívható, például: *a.~Vektor()*.)

Míg objektumtömb esetén az elemek konstruktorai az indexek növekvő sorrendjében hívódnak meg, addig a destruktort hívások sorrendje fordított. Az elmondottak dinamikus helyfoglalású tömbökre csak akkor érvényesek, ha a **delete[]** operátort alkalmazzuk a tömb megszűntetésére:

```
Vector * vt;
vt = new Vector[5];
// ...
delete [] vt;
```

Amennyiben a tömb törleszt (helytelenül) a **delete** művelettel végezzük, csak a 0. tömbelem destruktora fut le:

```
delete vt;
```

A destruktorkok használatához szintén egy sor, a konstruktoroknál említett szabály kö-

- Neve a hullám (~) karaktert követően megegyezik az őt tartalmazó osztály nevével.
- Ha nem készítenénk destruktort, akkor a fordító az alapértelmezettet használja.
- Nincs visszatérési értéke (**void** típusú sem).
- Deklarációjában nem szerepelhetnek a **static**, **const**, **mutable** és **volatile** kulcsszavakat.
- Nem készíthető mutató a destruktorkhoz.

A destruktorkok azonban több pontban különböznek a konstruktoroktól:

- Nincsenek paraméterei.
- Nem definiálható át (nem terhelhető túl), azonban az öröklés során lecserelehető.
- A destruktork virtuális (virtual) tehető. Ha az alaposztály destruktora virtuális, akkor a származtatott osztály destruktora is automatikusan virtuális lesz.
- Explicit módon is hívható.

14. Operátorok túlterhelése (operator overloading)

A C++ nyelv biztosítja annak a lehetőséget, hogy valamely, programozó által definiált függvényt szabványos operátorhoz kapcsoljunk, kibővíve ezzel az operátor működését. Ez a függvény automatikusan meghívódik, amikor az operátort egy meghatározott szöveggkörnyezetben használjuk.

Operátortfüggvényt azonban csak akkor készíthetünk, ha annak legalább egyik paramétere osztály (**class**, **struct**) típusú. Ez azt jelenti, hogy a **void** függvények, illetve a csak alap adattípusú argumentumokat használó függvények nem lehetnek operátortfüggvények. Az operátortfüggvény deklarációjának formája:

```
tipus operátor op(paraméterlista);
```

ahol az *op* helyén az alábbi C++ operátorok valamelyike állhat:

[]	()	.	->	++	--	&	new
*	+	-	~	!	/	%	new[]
<<	>>	<	>	<=	>=	==	delete
=	^		&&		=	*	delete[]
/=	%=	+=	-=	<<=	>>=	&=	
~=	=	,	->*				

Nem definiálhatók át a tagkiválasztás (**.**), az indirekt tagkiválasztás (*****), a hatókörfeloldás (**::**), a feltételes (**?:**), a **sizeof** és a **typeid** operátorok. Felhívjuk a figyelmet arra, hogy az operátorok túlterhelésével nem változtatható meg az operátorok elsőbbsége (precedenciája) és csoportosítása (asszociativitása), valamint nincs mód új műveletek bevezetésére sem. Az operátortfüggvények többségének visszatérési értéke és típusa azonban szabadon megválasztható.

Kifejezés	Művelet (@)	Tagfüggvény	Külső függvény
-----------	-------------	-------------	----------------

ga	++	A::operator@()	operator@ (A)
ga	+- * & ! ~ ++		
a@	++ --	A::operator@ (int)	operator@ (A, int)
a@b	+ - * / % ^ &		
a@b	< > == != <= >=	A::operator@ (B)	operator@ (A, B)
a@b	= += -= *= /=		
a@b	% = ^ = & = = <<= >>=	A::operator@ (B)	
a (b, c, ...)	()	A::operator@ () (B, C, ...)	
a->b	->	A::operator->()	

Az operátorfüggvényeket általában osztályon belül adjuk meg, a felhasználói típus le-
hetőségünk kiterjesztése céljából. Az =, (), [] és -> operátorokat azonban csak nem
statikus tagfüggvényekkel lehet átdefiniálni. A new és a delete operátorok esetén a túl-
terhelés statikus tagfüggvényekkel történik. Minden más operátorfüggvény elkészíthető
tagfüggvényként vagy az osztály külső (friend) függvényeként.

Az elmondottakat jól szemlélteti az előző oldal alján elhelyezett táblázat, ahol cso-
portba rendeztük a C++ nyelv átdefiniálható műveleteit, a dinamikus memóriake-
zelés operátorainak elhagyásával. A táblázatban a @ a műveleti jelet helyettesíti, míg
a, b és c valamely A, B és C osztály objektumai.

A túlterhelhető C++ műveleti jelek többségét (az előző táblázat @ operátorait) az
operandusok száma alapján két csoportra oszthatjuk. Erre a két esetre az alábbi
táblázatban összefoglaltuk az operátorfüggvények hívásának formáját.

Kétooperandusú operátor esetén:

Megvalósítás	Szintaxis	Aktuális hívás
tagfüggvény	$X \text{ op } Y$	$X.\text{operator op}(Y)$
friend függvény	$X \text{ op } Y$	$\text{operator op}(X, Y)$

Egyoperandusú operátor esetén:

Megvalósítás	Szintaxis	Aktuális hívás
tagfüggvény	$\text{op } X$	$X.\text{operator op}()$
tagfüggvény	$X \text{ op}$	$X.\text{operator op}()$
friend függvény	$\text{op } X$	$\text{operator op}(X)$
friend függvény	$X \text{ op}$	$\text{operator op}(X, 0)$

Bizonyos műveleteket átdefiniálása során a szokásostól eltérő megfontolásokra is
szükség van. Ezen operátoroknak – a Vektor példát követően – egy-egy alfejezetet
szenteltünk a könyvben.

Példaként tekintsük a Vektor osztály bővített változatát, amelyben túlterheljük az
indexelés ([]) , az értékadás (=) és az összeadás (+, +=) műveletfüggvényeket! Az ér-
tékdadás megvalósítására a több elemeknek másolása érdekében volt szükség. A + ope-
rátor barátfüggvényekkel valósfűjük meg, mivel a keletkező Vektor logikailag egyik
operandushoz sem tartozik. Ezzel szemben a += művelet megvalósításához tagfügg-
vényt használunk. Az osztály teljes deklarációját a Vektor.h, a definícióját pedig a
Vektor.cpp állomány tartalmazza.


```
// Vektor.h

class Vektor {
public:
    Vektor();
    Vektor(int n);
    Vektor(const Vektor& v);
    Vektor(const int a[], int n);
    ~Vektor() { delete[] p; }
    int fh() const { return merei; }
    int& operator[] (int i);
    Vektor operator= (const Vektor& v); // az = operátor
    Vektor operator+= (const Vektor& v); // a += operátor
    // a + operátor
    friend Vektor operator+ (const Vektor& v1,
                             const Vektor& v2);

private:
    int *p;
    int merei;
};

// Vektor.cpp (részlet)

int& Vektor::operator[] (int i) {
    if (i < 0 || i > merei-1) // indexhatár-ellenőrzés
        throw i;
    return p[i];
}

Vektor Vektor::operator= (const Vektor& v) {
    delete [] p;
    p=new int [merei=v.merei];
    for (int i = 0; i < merei; ++i)
        p[i] = v.p[i];
    return *this;
}

Vektor Vektor::operator+ (const Vektor& v) {
    int m = (merei < v.merei) ? merei : v.merei;
    for (int i = 0; i < m; ++i)
        p[i] += v.p[i];
    return *this;
}

Vektor operator+ (const Vektor& v1, const Vektor& v2) {
    Vektor sum(v1);
    sum+=v2;
    return sum;
}
```

A *Vektor* osztály felhasználását az alábbi programrészlet szemlélteti:

```
#include <iostream>
using namespace std;
#include "Vektor.h"

void show(Vektor & v) {
    for (int i=0; i<v.Fh(); i++)
        cout<<v[i]<<'\t';
    cout<<endl;
}

void main() {
    int a[]={4,9}, b[]={12,23,79};
    Vektor x(a,2),y(b,3),z;
    try {
        show(x); // 4      23      79
        show(y); // 12     23
        x=y;
        show(x); // 12     23
        x=Vektor(a,2);
        show(x); // 4      23
        x+=y;
        show(x); // 16     32
        z=x+y;
        show(z); // 28     55
    }
    catch (int n) {
        cout<<"Hibas tombindex: "<<n<<endl;
    }
    cin.get();
}
```

14.1 A lépítő (++ , --) operátorok túlterhelése

Külön kell szólnunk a lépítő operátorok túlterheléséről, hiszen ugyanazt a műveletet lehet használnunk előrevetett és hátravetett formában. A kérdés az, hogyan különböztethetjük meg ezt a két esetet, hiszen az operátorfüggvények neve megegyezik. A választ magától értetődik, különböző paraméterekkel véggezzük az átdefiniálást.

Mielőtt elmerülnénk a részletekbe, ismerkedjünk meg a függvények látszólagos (*dummy*) paraméterezésével! Bizonyos esetekben, például egy jövőbeni továbbfejlesztés, vagy egyszerűen a túlterhelés érdekében bizonyos paramétereket nem használunk a függvényben, bár a paraméter sorban szerepelnek. Ezen *dummy* paraméterek helyét a típusokkal jelöljük ki.

```
void vanincs(int a, long, char * an)
{
    cout<< an<<" erteke: "<< a<<endl;
}
```

Az ilyen függvényeket a szokásos módon, az argumentumok felsorolásával hívjuk. A látszólagos paraméternek megfelelő argumentum nem hagyható el, sőt kompatibilis típusúnak kell lennie:

tipusúnak kell lennie:

```
int nt=1223;
vannings(nt, 2005, "nt");
```

A fejezet elején megadott táblázatokból láthatjuk, hogy a fordító egy **int** típusú látszólagos paraméter segítségével küldönbözeti meg a léptető operátorok prefixes és postfixes alkalmazását. Ennek ismeretében a szűkséges függvények minden további nélkül elkészíthetők.

Az alábbi példában az előrevetett ++ operátor 10-el, míg a hátravetett 100-zal növeli meg a *Leptio* objektum adattagjának értékét:

meg a *Lepieto* objektum adattagjának értékét:

```

class Leppegto {
public:
    Leppegto (int _a=0) { a = _a; }
    friend int operator ++ (Leppegto &, int);
    int operator ++() { return a += 10; }
    int geta() { return a; }
};

int operator ++ (Leppegto & d, int)
{
    return d.a += 100;
}

void main()
{
    Leppegto n(12);
    cout<<"+n + n++<<endl;
    // 22 + 122 = 144
    cout<<n.geta()<<endl;
    // 122

    Leppegto m(12);
    cout<<m++ + ++m<<endl;
    // 112 + 122 = 234
    cout<<m.geta()<<endl;
    // 122
}

```

Armenyiben csak az egyik alakú (prefix vagy postfix) létező operátort készítyük el az osztályunkhoz, bizonyos fordítók a másik alakú létezés esetén is hajlandók a definiált operátort alkalmazni. Ezt a fordítás során egy figyelmeztető üzenet hozza tudtunkra.

14.2 Az értékadó operátor átdefinálása

Már ismert számunkra, hogy a C++ osztályok automatikusan rendelkeznek az alapértelmezett (paraméter nélkül) és a másoló konstruktorral, az objektumpéldányok közötti értékadást megvalósító művelettel valamint egy destruktortal. Amennyiben valamilyen konstruktor készítettünk, az alapértelmezettet is meg kell valósítanunk. Saját másoló konstruktor, értékadó operátor és destruktort akkor kell készítenünk, ha az osztály példányaihoz dinamikus memóriaterület tartozik.

Az alábbi példában szereplő *CopyCls* osztály objektumai egy életkort és egy nevet tartalmaznak. A név tárolásához dinamikusan foglalunk memóriaterületet, melyet a destruktortban fel is szabadítottunk.

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

class CopyCls {
private:
    int kor;
    char * nev;
    // Memóriafoglalás és a karaktersorozat bemásolása
    void Masol(int k, char * p) {
        kor = k, nev=NULL;
        if (p) {
            nev = new char[strlen(p)+1]; assert(nev);
            strcpy(nev, p);
        }
    }
public:
    // Alapértelmezett konstruktor
    CopyCls (int k=0, char * n=NULL) {
        Masol(k, n);
    }
    // Másoló konstruktor
    CopyCls (const CopyCls & cc) {
        Masol(cc.kor, cc.nev);
    }
    // Értékadó operátor
    CopyCls operator = (const CopyCls & cc) {
        // Ha nem önmagára vonatkozik az értékadás
        if (&cc != this) {
            delete[] nev, nev=NULL;
            Masol(cc.kor, cc.nev);
        }
        return *this;
    }
}
```

```

~CopyCls() { delete[] nev; }
int GetAdat() { return kor; }
char *GetNev() { return nev; }
};

void main()
{
    CopyCls a(25, "Lafentita", b(15, "Anna"), c(b), d,
    c = c;
    d = c = a;
    cout<<c.GetAdat()<<endl;
    cout<<c.GetNev()<<endl;
    cin.get();
}

```

A legfontosabb konstruálási műveleteken túlmenően, az értékadó operátor megvalósítására hívjuk fel a figyelmet. A függvényben csak akkor foglalnunk új területet és dobjuk el a régi, ha nem önmagára vonatkozik az értékadás. Az értékadó kifejezés értéke egy ideiglenes, a második konstruktorral létrehozott objektumpéldány, melyet az operátorfüggvény ad vissza az aktuális objektum másolataként. (Így működik az értékadás a szabványos C++ típusú operandusokkal.) A megoldás gyorsabbá tehető, ha kiküszöböljük az ideiglenes objektum létrehozását. Ehhez az operátorfüggvény visszatérési típusát *CopyCls &* (vagy *const CopyCls &*) hivatkozási típusúra kell változtatnunk.

14.3 Függvényeszerű osztályok (*functionoidok*) készítése

Az objektum-orientált C++ nyelv lehetővé teszi, hogy az eddig függvényekkel megvalósított megoldásokat függvényeszerű objektumokkal (*functionoidokkal*) helyettesítsük. A függvényhívás operátorának átdefinálásával a hívás formája sem változik. Bizonyos függvények a működése az előző állapotok ismeretére épül (emlékező algoritmusok.) Az emlékezőst statikus élettartamú lokális, vagy globális változók segítségével valósíthatjuk meg. Tipikus példa erre a Fibonacci-sorozat soron következő elemével visszatérő *fib()* függvény, amelynél azonban gondot jelent a sorozat újra-írdítása:

```

unsigned fibo()
{
    static unsigned a0=0, a1=1;
    unsigned a2;
    a2=a0+a1, a0=a1, a1=a2;
    return a2;
}

```

A *Fibonacci* osztályban az adatok és a műveletek összeépítése egy magasabb szintű, használhatóbb megoldáshoz vezet. Külön felhívjuk a figyelmet a függvényhívás operátorának túlterhelésére és a sorozat újraindításának megoldására.

```

class Fibonacci {
protected:
    unsigned a0, a1;
public:
    Fibonacci(): a0(0), a1(1) {}
    void Restart() { a0=0; a1=1; }
    int operator() () {
        unsigned a2;
        a2=a0+a1, a0=a1, a1=a2;
        return a2;
    };
    void main()
    {
        Fibonacci Fibo;
        for (int i=1; i<12; i++)
            cout<<Fibo()<<" ";
        Fibo.Restart();
        for (int i=1; i<12; i++)
            cout<<Fibo()<<" ";
    }
}

```

Az átdefiniált **operator()** függvény másik felhasználási területe az ún. *iterátorok* készítése, melyek segítségével könnyelműen elérhetjük az adott osztály objektumaiban tárolt adatokat. Erre látunk példát a fejezetet záró *string*c osztály bővített változatában.

14.4 Felhasználó által definiált típuskonverzió

A C++ nyelv támogatja, hogy a programozó a saját adattípusához (osztály) típuskonverziókat rendeljen. Az ilyen felhasználó által definiált típuskonverziót végrehajtó tagfüggvény deklarációja:

```
operator típus() ;
```

A függvény visszatérési értékének típusa megegyezik a függvény nevében használt típussal. A típuskonverziós operátor csak visszatérési típus és argumentumlista nélküli tagfüggvény lehet.

Az alábbi példában a komplex típust osztályként valósítjuk meg. Az egyetlen nem **double** típusú argumentummal rendelkező konstruktor elvégzi a más típusról – a példában **double** – **cplx** típusra történő konverziót. A fordított irányú átalakításhoz **double** nevű konverziós operátort definiáltunk.

```

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

class cplx {
public:
    cplx () { re=lm=0; }
    cplx(double a) { re=a; lm=0; } // konverziós konstruktor
    cplx(double a, double b) { re=a; lm=b; } // konverziós konstruktor
private:
    operator double() { return sqrt(re*re+lm*lm); }
    double re, lm;
};

void main() {
    cplx a(3,4);
    cout << double(a)<< endl; // a kiírt érték: 5
    cout << double(cplx(10))<< endl; // a kiírt érték: 10
}

```

Megjegyezzük, hogy a *cplx* osztály három konstruktora egyetlen konstruktorral helyettesíthető, amelyben alapértelmezett paramétereket használunk:

```
cplx(double a=0, double b=0) { re=a; lm=b; }
```

14.5 Az osztályok bővítése input/output műveletekkel

A C++ nyelv lehetővé teszi, hogy az osztályokon alapuló adatfolyamoknak „megtanítsuk” a saját készítésű osztályunk objektumainak kezelését. Az adatfolyamosztályok közül az *istream* az adatebvitelért, míg az *ostream* az adatkivitelért felelős. Az input/output műveletek végzéséhez pedig az átdefiniált $>>$ és $<<$ operátorokat használjuk. A szükséges működés eléréséhez *friend* operátorfüggvényként el kell készítenünk a fenti műveletek saját átdefiniált változatát, mint ahogy az a *cplx* osztály bővített változatában látható:

```

#include <cmath>
#include <csdlo>
#include <iostream>
using namespace std;

class cplx {
public:
    cplx(double a=0, double b=0) { re=a; lm=b; }
    operator double() { return sqrt(re*re+lm*lm); }
    friend istream & operator<<>(istream &, cplx &);
    friend ostream & operator<<>(ostream &, const cplx &);
};

```

```
// Az adatbeviteli formátuma: 12.23+7.79i, illetve 12.23-7.79i
istream & operator>>(istream & is, cplx & c) {
    char p[256];
    is.getline(p, 256);
    if (sscanf(p, "%lf%lf", &c.re, &c.im) != 2)
        c=cplx(0);
    return is;
}

// Adatkiviteli formátum: 12.23+7.79i, illetve 12.23-7.79i
ostream & operator<<(ostream & os, const cplx & c) {
    os<<c.re<<("c.im<0? '-' : '+')<<abs(c.im)<<".i";
    return os;
}

void main() {
    cplx a, b;
    cout<<"Kérek egy komplex számot: ";
    cin >>a;
    cout<<"A komplex szám: " <<a<<endl;
    cin.get();
}
```

14.6 A new és a delete operátorok túlterhelése

A **new** és a **delete** operátorok átdefiníálásokor további megköötéseket kell szem előtt tartanunk. A **new** operátor osztálytag operátorfüggvényét **void * visszatérési értékkel** és **size_t** típusú első paraméterrel kell definiálni. Ebben az argumentumban a fordító automatikusan átadja az osztály bájttban kifejezett méretét.

A **delete** operátor esetén, az osztálytag operátorfüggvényét **void * első és size_t** típusú opcionális második paraméterrel kell létrehozunk. Az utóbbi paramétert, amennyiben megadtuk, a fordító használja az első paraméter által megcímzett objektum méretének átadására.

Az átdefiniált **new** és **delete** operátorok mellett, a hatókör operátor megadásával mindig elérhetjük az eredeti **new** és **delete** műveleteket:

```
char *p=new char[1001];
::delete p;
```

A **new** és **delete** operátorok automatikusan az őket definiáló osztály statikus tagjaként kerülnek lefordításra. Ebből következik, hogy a **this** mutató nem használható, így az operátorfüggvényekből csak a statikus adattagokat érhetjük el. Ez azért szükséges, mivel a **new** és **delete** hívását az osztályobjektum létezése nélkül is végre kell tudni hajtani.

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



```

stringc(char) ;
stringc(const char *) ;
stringc(const stringc &) ;
~stringc(void) {delete[] p;}
char* operator[] (int) ;
// index operator
stringc operator = (stringc) ;
// == operator
bool operator == (stringc) ;
// == operator
operator char * () const {return p;} // stringc->char * konv.
operator int () {return h;} // a string hossza
stringc operator + (stringc) ; // + operátorok
friend stringc operator+(stringc, char) ;
friend stringc operator+(char, stringc) ;

```

// Az osztály deklarácója

```

class stringc
public:
    using namespace std;
    #include <istream>
    #include <cassert>
    #include <cstring>

```

Az eddigi ismeretünk elegendőek ahhoz, hogy a C++ nyelvet saját adattípussal bővítsük. Az adattípushoz osztályt kell készítenünk megfelelő adattagokkal, majd gondoskodnunk kell a különféle konstruktorai lehetőségekről. Ezt követően a szükséges operátorok túlterhelésével a típusunk ugyanúgy használható, mint a nyelv alapítípusai. Példaként tekintsünk a C-karaktertorsorozatokra lefedő *stringc* osztályt!

14.7 A C++ nyelv bővítése saját típussal

```

}
cin.get();
delete p;
p->d=1223;
osztaly *p=new osztaly;

void main()
{
}

void osztaly::operator delete(void *p, size_t n) {
    cout<<"Torles"<<endl;
    ::delete static_cast<osztaly *>(p);
}

void * osztaly::operator new(size_t n) {
    cout<<"Foglalas:"<<n<<endl;
    return ::new char[n];
}

void operator delete(void *p, size_t n);
void * operator new(size_t n);
double d;
public:
class osztaly {

```

Foglalas:8
Torles

```

// l/o műveletek
friend ostream& operator<<(ostream& , const string& &);
friend ostream& operator>>(istream& , string& &);
private:
    char* p;
    int h;
    string&(int);
};

// Az osztály definíciója
// Konstruktorok
string&::string&() {
    h = 0;
    p = new char[1]; assert(p!=0);
    *p = 0;
}

string&::string&(char c) {
    h = 1;
    p = new char[2]; assert(p!=0);
    p[0] = c;
    p[1] = 0;
}

string&::string&(const char* s) {
    h = strlen(s);
    p = new char[h+1]; assert(p!=0);
    strcpy(p, s);
}

string&::string&(const string& s) {
    h = s.h;
    p = new char[h+1]; assert(p!=0);
    strcpy(p, s.p);
}

string&::string&(int n) {
    p = new char[n+1]; assert(p!=0);
    p[0] = 0;
    h = n;
}

// Operator-tagüggvények
char& string&::operator[](int i) {
    if (i < 0 || i > h)
        throw "indexhatar tullepes!\n";
    return p[i];
}

bool string&::operator==(string& s) {
    return strcmp(p, s.p);
}

```

```

stringc stringc :: operator=(stringc s) {
    if ( this == &s )
        return *this;
    delete p;
    h = s.h;
    p = new char[h+1]; assert(p!=0);
    strcpy(p, s.p);
    return *this;
}

stringc stringc :: operator + (stringc s) {
    int th = h + s.h;
    stringc ts(th);
    strcpy(ts.p, p);
    strcat(ts.p, s.p);
    return ts;
}

stringc operator + (stringc s, char c) {
    stringc cs(c);
    return s + cs;
}

stringc operator + (char c, stringc s) {
    stringc cs ( c );
    return cs + s;
}

ostream& operator<<(ostream& out, const stringc & s) {
    out<<s.p;
    return out;
}

istream& operator>>(istream& in, stringc & s) {
    char cp[256];
    in.getline(cp,256);
    s=stringc(cp);
    return in;
}

// Az osztály használatát bemutató főprogram
void main() {
    stringc s1;
    stringc s2("Lafénitá");
    stringc s3 = s2;
    stringc s4 = "C# nyelv";
    cout<<"Kerek egy neve: ";
    cin>>s1;
    cout<<s1<<endl;
    s3="Strousstrup";
    s1 = s3;
}

```

```

s4[1] = s4[2] = '+';
cout<<s4<<endl;
if ( s1 == s3 ) {
    s2 = s1 + stringc(" & " ) + s4;
    s3 = s2 + '\n';
    cout<<s3;
}
s2 = '{' + s2 + '}' ;
cout << s2<<endl;
char *sp=new char[s2];
strcpy(sp,s2);
cout<<s2<<endl;
cin.get();
}

```

A *main()* függvény futásának eredményét az alábbiakban láthatjuk

```

Kerek egy nevet: Bjärne Stroustrup
Bjärne Stroustrup
C++ nyelv
Stroustrup & C++ nyelv
Stroustrup & C++ nyelv
Stroustrup & C++ nyelv

```

Gyakran használjuk a függvényhívás operátorának átdefiniált változatát, az ún. *iterátor* tagfüggvény elkészítéséhez. Az iterátor segítségével egymás után elérhetjük a karaktertorsorozatok karaktereit.

A *string* osztály bővítése után

```

char operator() () ;
// iterátor

char stringc :: operator () () {
    static int index=0;
    if (index<h)
        return p[index++];
    else
        return (index=0);
}

```

az alábbi programcszettel egymás alatt jelenítjük meg az *s4* string karaktereit:

```

char ch;
do
    cout<<(ch=s4())<<endl;
while (ch);

```

15. Az öröklés (öröklődés) mechanizmusa

Az előző fejezetekben megismertük, miként kell egymástól független osztályokat létrehozni, és azokat a C++ nyelv szerves részévé tenni. Az OOP adta lehetőségek azonban nem merülnek ki ebben. A problémák objektum-orientált megközelítésben általában egy új programépítési módszert, a származtatást (öröklés, *inheritance*) alkalmazunk. Az osztályok származtatása lehetővé teszi, hogy már meglévő típusok adatait és műveleteit újra felhasználjuk, illetve a feladat igényeinek megfelelően módosítsuk, bővítsük. A problémákat így nem egyetlen (nagy) osztállyal, hanem osztályok egymásra épülő rendszerével (hierarchiájával) oldjuk meg.

Az öröklés (*inheritance*) az objektum-orientált C++ nyelv legfőbb sajátossága. Ez a mechanizmus teszi lehetővé, hogy bizonyos osztályokból más osztályokat származtassunk, mely osztályok a származtatás során adattagokat és tagfüggvényeket örökölnek. Az örökölt tulajdonságok tetszőlegesen kiterjeszthetők és megváltoztathatók. A C++ támogatja a többszörös öröklődést (*multiple inheritance*), melynek során valamilyen új osztályt több alaposztályból származtatunk.

15.1 A származtatott osztályok

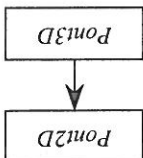
A származtatott osztály (*derived class*, utódosztály) olyan osztály, amely az adattagjait és a tagfüggvényeit egy vagy több előzőleg definiált osztálytól örököli. Azt az osztályt, amelytől a származtatott osztály szintén lehet alaposztály a további osztályoknak, lehetővé téve ezzel osztály-hierarchia kialakítását.

A származtatott osztály az alaposztály minden tagját örököli, de az alaposztályból csak a **public** és **protected** tagokat éri el sajátjaként. A származtatott osztályban az örökölt tagokat saját adattagokkal és tagfüggvényekkel egészíthetjük ki. A származtatás kijelölésére az osztály fejlét használjuk:

```
class Szarmaztatott : public Alap1, ...private Alap2
{
    // az osztály törzse
};
```

Az alaposztálybeli elérhetőségtől függetlenül nem öröklődnek a konstruktorok, a destruktork, az értékadó operátor, illetve a *friend* deklarációk. Az esetek többségében nyilvános öröklést használunk, mivel ekkor az utódobjektum minden kontextusban helyettesítheti az ősobjektumot.

Az alábbi – síkbeli és térbeli pontokat definiáló – példában bemutatjuk a származtatás alkalmazását. A létrehozott osztály-hierarchia igen egyszerű:



```

class Pont2D {
protected:
    int x,y;
public:
    Pont2D(int a=0, int b=0) { x=a; y=b; }
    Pont2D(Pont2D & p) { *this = p; }
    void SetPont2D(int a=0, int b=0) { x=a; y=b; }
    void GetPont2D(int& a, int& b) { a=x; b=y; }
};

```

```

class Pont3D : public Pont2D {
protected:
    int z;
public:

```

```

    Pont3D(int a=0, int b=0, int c=0):Pont2D(a,b),z(c) {}
    Pont3D(Pont3D & p):Pont2D(p.x, p.y),z(p.z) {}
    void SetPont3D(int a=0, int b=0, int c=0) { x=a; y=b; z=c; }
    void GetPont3D(int& a, int& b, int& c) { a=x; b=y; c=z; }
    void Kíír() { cout<<'('<<x<<','<<y<<','<<z<<')'<<endl; }
};

```

```

void Megjelenít(Pont2D & p) {
    p.Kíír();
}

```

```

void main()
{
    Pont2D p1(12,23), p2(p1), p3;
    Pont3D q1(12,23,79), q2(q1), q3;
    p1.Kíír();
    q1.Kíír();
    q2.SetPont3D(1,2,3);
    p3=q2;
    p2.Kíír();
    Megjelenít(p3);
    Megjelenít(q3);
    cin.get();
}

```

(12, 23)
(12, 23, 79)
(12, 23)
(1, 2)
(0, 0)

Láthatjuk, hogy a **public** származtatással létrehozott osztály minden esetben (értékadás, függvényargumentum,...) helyettesítheti az alaposztályt. Ezen nem kell csodálkozni, hisz a származtatott osztály magában foglalja az alaposztályt.

A származtatási listában megadott **public**, **protected** és **private** kulcsszavak az öröklött (*nyilvános* és *védett*) tagok új osztálybeli elérhetőségét szabályozzák, az alábbi táblázatban összefoglalt módon.

Az öröklés módja	Alaposztálybeli elérés	Hozzáférés a származtatott osztályban	
		public	protected
public	<i>public</i>	<i>public</i>	<i>public</i>
	<i>protected</i>	<i>protected</i>	<i>protected</i>
protected	<i>private</i>	<i>public</i>	<i>protected</i>
	<i>protected</i>	<i>protected</i>	<i>protected</i>
private	<i>private</i>	<i>public</i>	<i>private</i>
	<i>protected</i>	<i>protected</i>	<i>private</i>

A **public** származtatás során az öröklött tagok megtartják az alaposztálybeli elérhetőségeiket, míg a **private** származtatás során az öröklött tagok a származtatott osztály privát tagjaivá válnak, így elzárjuk azokat mind az új osztály felhasználói, mind pedig a továbbfejlesztői elől. Védett (**protected**) öröklés esetén az öröklött tagok védett tagok lesznek az új osztályban, így továbböröklésük biztosított marad. (Az alapértelmezés szerinti származtatási mód **class** típusú alaposztály esetén a **private**, míg a **struct** típust használva a **public**.)

A friend viszony az öröklés során

Az alaposztály barátja (**friend**) a származtatott osztályban csak az alaposztályból öröklött tagokat érheti el. A származtatott osztály barátja (**friend**) az alaposztályból csak a **public** és a **protected** tagokat érheti el.

Az öröklött tagok elérése

Alapvetően a származtatott osztály öröklött tagjai ugyanúgy érhetők el, mint a saját tagok. Azonban elképzelhető olyan eset, amikor az öröklött tagok elérése akadályba ütközik. Ha származtatott osztály azonos névvel újradefiniálja az öröklött tagot, az láthatatlanná válik. Ilyen esetben a hatókör operátort kell használnunk a hivatkozáshoz:

Alap::tag

Arra is van lehetőség, hogy az öröklött tagok elérhetőségét egyedileg szabályozzuk. Ehhez az *Alap::tag*; hivatkozást a származtatott osztály megfelelő hozzáféréssel rendelkező részébe kell helyeznünk. Azonban vigyünk figyelembe, hogy az öröklött tag elérhetősége nem lehet nagyobb, mint amilyen az alaposztályban volt! Nézzünk egy példát az elmondottak szemléltetésére!

```
// Az alapszály
class B {
    int x, y;
public:
    int b1, b2;
    int Bfunc1(void) { return x; }
    int Bfunc2(void) { return y; }
};
```

```
// A származtatott osztály
class D: private B {
    // az öröklött b2 elfedése
    int b2;
    int d;
public:
    // a private származtatással öröklött b1 public elérésű lesz
    B: b1;
    // az öröklött Bfunc1() elfedése
    void Bfunc1(void);
};
```

```
void D::Bfunc1(void) {
    d = B::Bfunc1(); // a nem látható Bfunc1() elérése
    b1 = Bfunc2(); // a Bfunc2() látható
    b2 = B::b2; // a nem látható b2 adattag elérése
}

void main() {
    D od;
    od.b1 = 23;
}
```

Az alábbi táblázatban összefoglaltuk, hogy mely osztály, milyen (általá elérhető) tagokkal rendelkezik:

<i>A B alapszály tagjai:</i>	<i>A D származtatott osztály tagjai</i>
private: x, y	private: B::b2, b2, d,
public: b1, b2, Bfunc1(), Bfunc2()	public: b1, Bfunc1()

Öröklés vagy beágyazás?

Valamely probléma objektum-orientált megoldása során mérlegelni kell, hogy az öröklés vagy a beágyazás segítségével jutunk-e pontosabb modelllhez. A döntés általában nem egyszerű, mi is csak a technika bemutatására szorítkozunk.

Az alábbi két példaprogramból jól láthatók a két megoldás közötti különbségek, illetve azonosságok.

<i>Megoldás örökléssel</i> <pre> class Pont { protected: int x,y; public: Pont(int a=0, int b=0) {x=a, y=b;} Pont(const Pont & p) {x=p.x, y=p.y;} int getX() {return x;} int getY() {return y;} }; class Kor { protected: int r; public: Pont p; // beágyazás Kor(int x=0, int y=0, int r=0) : p(x, y), r(r) {} Kor(const Pont & p, int r=0) : p(p), r(r) {} int getr() {return r;} }; void main() { Kor k1(100, 200, 23); cout<<k1.p.getX()<<endl; cout<<k1.getY()<<endl; cin.get(); } </pre>	<i>Megoldás örökléssel</i> <pre> class Kor : public Pont { // öröklés protected: int r; public: Kor(int x=0, int y=0, int r=0) : Pont(x, y), r(r) {} Kor(const Pont & p, int r=0) : Pont(p), r(r) {} int getr() {return r;} }; void main() { Kor k1(100, 200, 23); cout<<k1.getX()<<endl; cout<<k1.getY()<<endl; cin.get(); } </pre>
---	--

15.2 Az alaposztály inicializálása

Az alaposztály(ok) inicializálására a taginicializációs lista kibővített változatát használjuk, amelyben a tagok mellett a közvetlen ős osztályok konstruktorait is felsoroljuk. Példaként tekintsük az *Alap1* és *Alap2* osztályokból származtatott *Szarvazatott* osztály konstruktorait!

```

#include <iostream>
using namespace std;

class Alap1 {
private:
    int na;
    long ta;
public:
    Alap1(int_na, long_ta) {na=_na; ta=_ta;}
    Alap1(Alap1 & nata) {*_this = nata;}
};

```

```

class Alap2 {
    private:
        char *pt;
        bool lv;
    public:
        Alap2(bool _lv, char * _pt) {lv=_lv; pt=_pt;}
        Alap2(Alap2 & lvpt) {*this = lvpt;}
};

```

```

class Szarmaztatott : public Alap1, public Alap2 {
    private:
        char ln;
    public:

```

```

        Szarmaztatott(int na, long ta, bool lv, char *pt, char ln);
        Szarmaztatott(Szarmaztatott & dc);
};

```

```

        Szarmaztatott::Szarmaztatott(int na, long ta,
        bool lv, char *pt, char ln) : Alap1(na, ta), Alap2(lv, pt)
        {
            // a származtatott osztály konstruktora
        }

```

```

        Szarmaztatott::Szarmaztatott(Szarmaztatott & dc)
        : Alap1(dc), Alap2(dc)
        {

```

```

            // a származtatott osztály konstruktora
        }

```

```

    void main() {
        Szarmaztatott d1(23, 1979, true, "Lafentita", 'N');
        Szarmaztatott d2(d1);
        cin.get();
    }

```

A *Származtatott* osztály példányosítása során a fordító az alábbi sorrendben hívja a konstruktorkat:

1. Végrehajlódnak az alaposztályok konstruktora az inicializációs lista szerinti sorrendben, balról-jobbra haladva.
2. Meghívódnak a származtatott osztály tagobjektumainak konstruktora, az objektumtagok megadásának sorrendjében (a példában nem szerepelnek).
3. Left az osztály saját konstruktora.

Az objektumpéldány megszűnésekor a destruktorkok a fentiekkel ellentétes sorrendben hajlódnak végre.

Az alaposztály inicializálása elvégezhető alaposztálytípusú objektummal, vagy a származtatott osztály objektumával, ha **public** módon származtatjuk az új osztályt. (A fenti példa második konstruktorkorában alkalmaztuk ezt a megoldást.) Az inicializálást ekkor a második konstruktorkor végzi.

15.3 Virtuális tagfüggvények – többalakúság (polimorfizmus)

A virtuális függvény olyan **public** vagy **protected** tagfüggvénye az alaposztálynak, amelyet a származtatott osztályban újradefiniálhatunk az osztály „viselkedésének” megváltoztatása (polimorfizmus) érdekében. A virtuális függvény általában a nyilvános alaposztály referenciáján keresztül hívódik meg, melynek aktuális értéke a program futása során alakul ki (dinamikus összerendelés, késői kötés).

Ahhoz, hogy egy tagfüggvény virtuális legyen, a **virtual** kulcsszót kell megadnunk a függvény deklarációja előtt:

```
class Pelda {
    public:
        virtual int pf();
};
```

Nem szükséges, hogy az alaposztályban a virtuális függvénynek a deklarációja is szerepeljen. Ebben az esetben ún. tisztán virtuális függvénnel (*pure virtual function*) van dolgunk:

```
class Pelda {
    public:
        virtual int pf()=0;
};
```

Egy vagy több tisztán virtuális függvényt tartalmazó osztállyal (*absztrakt osztály*) nem készíthetünk objektum példányt. Az absztrakt osztály csak az öröklés kiinduló pontjaként, alaposztályaként használható.

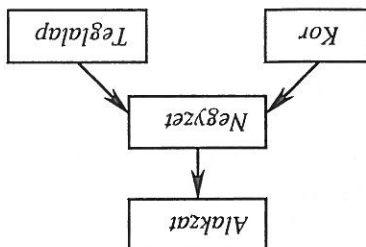
15.3.1 A virtuális függvények újradefiniálása (redefine)

Ha egy függvényt az alaposztályban virtuálisaként deklaráltunk, akkor ezt a tulajdonságát változtatva újradefiniálhatjuk, de az öröklött verziót is használhatjuk. Saját verzió definíciójaként nem szükséges a **virtual** szót megadnunk.

Ha egy származtatott osztály tisztán virtuális függvényt örökölt, akkor ezt mindenképpen saját verzióval kell újradefiniálni, mivel különben a származtatott osztály is absztrakt osztály lesz. A származtatott osztály tartalmazhat olyan virtuális függvényeket is, amelyeket nem az alaposztálytól örökölt.

A származtatott osztályban a virtuális függvény újradefiniált változatának pontosan (név, típus, paraméterlista) meg kell egyeznie az alaposztályban definiálttal. Ha a két deklaráció nem pontosan egyezik, akkor az újradefiniálás helyett a túlterhelés (*overloading*) mechanizmusa érvényesül.

Az alábbi példaprogramban mindegyik alakzat saját maga számolja ki a területét és a kerületét, azonban a megjeleltést az absztrakt alapsztály (*Alakzat*) végzi. Az osztályok hierarchiája az ábrán látható:



```

#include <iostream>
using namespace std;

// Absztrakt alapsztály
class Alakzat {
protected:
    int x, y;
public:
    Alakzat(int _x=0, int _y=0) {x=_x; y=_y;}
    virtual double terület()=0;
    virtual double kerület()=0;
    void megjelenit() {
        cout<<'('<<x<<','<<y<<')\t';
        cout<<"\tTerület: "<<terület();
        cout<<"\tkerület: "<<kerület();
    }
};

class Negyzet : public Alakzat {
protected:
    double a;
public:
    Negyzet(int _x=0, int _y=0, double _a=0)
        : Alakzat(_x,_y) {a=_a;}
    double terület() {return a*a;}
    double kerület() {return 4*a;}
};

class Teglalap : public Negyzet {
protected:
    double b;
public:
    Teglalap(int _x=0, int _y=0, double _a=0, double _b=0)
        : Negyzet(_x,_y,_a) { b=_b; }
    double terület() {return a*b;}
    double kerület() {return 2*(a+b);}
};

```

```

class Kor : public Negyzet {
    static const double pi;
public:
    Kor(int _x=0, int _y=0, double _a=0)
        : Negyzet(_x,_y,_a) {}
    double terület() {return a*pi;}
    double kerület() {return 2*a*pi;}
};

const double Kor::pi=3.14159265;

void main() {
    Negyzet n(12,23,10);
    cout<<"Negyzet: ";
    n.megjelenit();
    Kor k(23,12,10);
    cout<<"Kor: ";
    k.megjelenit();
    Teglalap t(12,7,10,20);
    cout<<"Teglalap: ";
    t.megjelenit();
    cin.get();
}

```

A virtuális függvények használata és a nyilvános öröklés lehetővé teszi, hogy a hierarchia minden objektumával hívható külső függvényeket hozzunk létre:

```

void megjelenit2(Alakzat & a) {
    cout<<"Terület: "<<a.terület()<<endl;
    cout<<"Kerület: "<<a.kerület()<<endl;
}

```

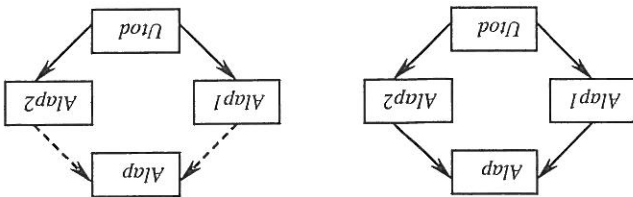
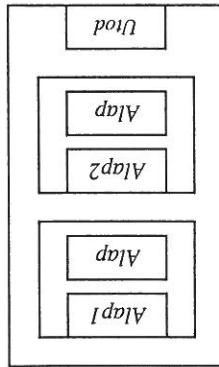
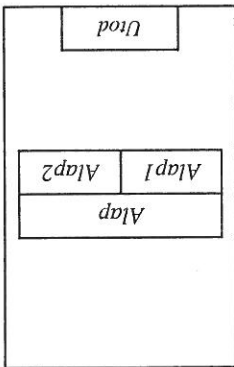
Virtuális tagfüggvényeket tartalmazó osztályok esetén a fordítóprogram az osztályhoz kapcsolódóan egy ún. virtuális táblát (VT) épít fel, amely az aktuális újradefiniált virtuális függvények címét tartalmazza. Az osztályhierarchiában található azonos nevű virtuális függvények azonos indexszel szerepelnek ezekben a táblákban, ami lehetővé teszi a virtuális tagfüggvények teljes lecserélését.

15.3.2 Virtuális destruktorkok

A destruktort virtuális függvényként is definiálhatjuk. Ha az alaposztály destruktora virtuális, akkor minden ebből származtatott osztály destruktora is virtuális lesz. Ezáltal biztosak lehetünk abban, hogy a megfelelő destruktort hívódik meg, amikor az objektum megszűnik.

15.4 Virtuális alaposztályok

A többszörös öröklődés során problémát jelenthet, ha ugyanazon alaposztály több példányban jelenik meg a származtatott osztályban. A virtuális alaposztályok használata-val az ilyen jellegű problémák kiküszöbölhetők.

az Alap nem virtuális
alaposztályaz Alap virtuális
alaposztály

```

#include <iostream>
using namespace std;

class Alap {
public:
    int q;
};

// az Alap virtuális alaposztály
class Alap1 : virtual public Alap {
public:
    Alap1(int i) { x=i; }
};

// az Alap virtuális alaposztály
class Alap2 : public virtual Alap {
public:
    Alap2(int i) : y(i) {}
};

```

```

class Utdod: public Alap1, public Alap2 {
    int a,b;
public:
    Utdod(int i,int j): Alap1(i*5),Alap2(j+1), a(i), b=j,{}
};

void main()
{
    Utdod ux(10,20);
    ux.Alap1::q=100;
    cout << ux.Alap2::q<<endl; // 100
    ux.Alap2::q=200;
    cout << ux.q<<endl;
    cin.get();
}

```

Az alaposztály a **virtual** öröklés hatására csak egyetlen példányban lesz jelen a származtatott osztályokban, függetlenül attól, hogy hányszor fordul elő az öröklődési láncban. A példában a virtuális alaposztály q adattagját öröklik az *Alap1* és az *Alap2* alaposztályok. A virtuálisítás miatt az *Alap* alaposztály egyetlen példányban szerepel, így az *Alap1::q* és az *Alap2::q* ugyanarra az adattagra hivatkoznak. A **virtual** szó használata nélkül az *Alap1::q* és az *Alap2::q* különböző adattagokat jelölnek.

15.5 Futás közbeni típusinformációk (RTTI) osztályok esetén

A különböző vizuális fejlesztőrendszerek futás közbeni típusinformációkat (*RunTime Type Information*) tárolnak az objektum példányok mellett. Ennek segítségével a futató rendszerre bízhatjuk az objektumok típusának azonosítását, így nem kell nekünk erre a célra adattagokat bevezetnünk. Az *RTTI* mechanizmus helyes működéséhez polimorf alaposztályt kell kialakítanunk, vagyis legalább egy virtuális tagfüggvényt el kell helyeznünk benne, és engedélyeznünk kell az *RTTI* tárolását. (Az engedélyezési lehetőséget általában a fordító beállításai között találjuk meg.) A mutatók és referenciák típusának azonosítására a **dynamic_cast** és a **typeid** műveleteket, míg a megfelelő típusátalakítás elvégzésére a **dynamic_cast** operátort használjuk, az alábbi módon:

```

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

class Os {
public:
    virtual int vf(){} // e nélkül nem tárolódik RTTI
};

class Utdod : public Os {
public:
    void fu() {cout<<"Utdod"<<endl;}
};

```

```

void main() {
    Utod * putod = new Utod;
    Os * pos = putod;

    const type_info& tios=typeid(pos); // a mutató type-info-ja
    const type_info& tiutod=typeid(*pos); // az Utod type-info-ja
    cout<<tios.name()<<endl;
    cout<<tiutod.name()<<endl;

    if (typeid(*pos) == typeid(Utod)) // Utod-ra mutat?
        dynamic_cast<Utod*>(pos)->fu();
    if (dynamic_cast<Utod*>(pos))
        dynamic_cast<Utod*>(pos)->fu();
}

```

A következő példaprogramban az elérési problémák akkor jelentkeznek, amikor osztályonként különböző tagokat szeretnénk elérni.

```

#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

class Allat {
protected:
    int labak;
    virtual string fajta()=0;
public:
    Allat(int n) {labak=n;}
    void Info() {
        cout<<"A(z) "<<fajta()<<"nak "
            <<labak<<" laba van."<<endl;
    }
};

class Hal : public Allat {
protected:
    string fajta() {return "hal";}
public:
    Hal(int n=0) : Allat(n) {}
    void Uszik() {cout<<"Uszik"<<endl;}
};

class Madar : public Allat {
protected:
    string fajta() {return "madar";}
public:
    Madar(int n=2) : Allat(n) {}
    void Repul() {cout<<"Repul"<<endl;}
};

```



```
class Emlős : public Allat {
public:
    string fajta() {return "emlős";}
    void Fut() {cout<<"Fut"<<endl;}
};
```

```
void main() {
    const int db=3;
    Allat * p[db];
    p[0]=new Hal;
    p[1]=new Madar;
    p[2]=new Emlős;
```

```
// RTTI nélkül is működő lekérdezés
for (int i=0; i<db; i++)
    p[i]->Info();
```

```
// RTTI-alapú feldolgozás
for (int i=0; i<db; i++)
```

```
if (dynamic_cast<Hal*>(p[i])) // Hal?
    dynamic_cast<Hal*>(p[i])->Uszik();
else
```

```
if (typeid(*p[i])==typeid(Madar)) // Madár?
    dynamic_cast<Madar*>(p[i])->Repul();
else
```

```
if (typeid(*p[i])==typeid(Emlős)) // Emlős?
    dynamic_cast<Emlős*>(p[i])->Fut();
for (int i=0; i<db; i++)
```

```
delete p[i];
}
```

Az összehasonlítás kedvéért szerepeljen itt a fenti feladat futás közbeni típusinformációkat nem alkalmazó változata. Ekkor az *efajta* adattagban tárolt numerikus konstansokkal azonosítjuk az osztályokat. Természetesen használhatunk volna az azonosításhoz a *fajta()* tagfüggvényt is, azonban ezzel sokkal hosszadalmasabb megoldáshoz jutottunk volna.

```
enum {HAL, MADAR, EMLÓS};
class Allat {
protected:
    int labak;
    virtual string fajta()=0;
public:
    const int efajta;
    Allat(int n, int ef) : labak(n), efajta(ef) {}
    void Info() {
        cout<<"A(z) "<<fajta()<<"nak "
        <<labak<<" labá van."<<endl;
    }
};
```

```

    }
    cin.get();
    delete p[i];
    for (int i=0; i<db; i++)
    }
    static_cast<Emlos*>(p[i])->Put(); break;
    case EMLoS:
    static_cast<Madar*>(p[i])->Repu(); break;
    case MADAR:
    static_cast<Hal*>(p[i])->Uszik(); break;
    case HAL:
    switch (p[i]-> fajta) {
        for (int i=0; i<db; i++)
        p[i]->Info();
        for (int i=0; i<db; i++)
        // RTTI nélküli lekérdezések
        Ailat * p[db] = { new Hal, new Madar, new Emlos };
        const int db=3;
        void main() {
        };
        void Put() {cout<<"Put"<<endl;}
        Emlos(int n=4) : Ailat(n,EMLoS) {}
        public:
        string fajta() {return "emlos";}
        protected:
        class Emlos : public Ailat {
        };
        void Repu() {cout<<"Repu"<<endl;}
        Madar(int n=2) : Ailat(n,MADAR) {}
        public:
        string fajta() {return "madar";}
        protected:
        class Madar : public Ailat {
        };
        void Uszik() {cout<<"Uszik"<<endl;}
        Hal(int n=0) : Ailat(n,HAL) {}
        public:
        string fajta() {return "hal";}
        protected:
        class Hal : public Ailat {

```

15.6 Kivételek és az osztályok

A kivételek kezelése sok esetben egyszerűsíti a programunk hibás állapotainak feldolgozását, hiszen elegendő egy helyen felkészülni a különböző hibás helyzetekre. A kivételeket a tagfüggvényekben belüli bárhol előidézhettünk, illetve felléphetnek. A konstruktor és a destruktorki kivételével, amikor valamelyik tagfüggvényben fellép egy kivétel, automatikusan meghívódik az objektum destruktora, mielőtt egy külső kivételkezelő programrészhez adódna a vezérlés. Természetesen a kivételt helyben a tagfüggvényben belüli is feldolgozhatjuk, és még akkor is dönthetünk arról, hogy továbbadjuk-e kívülre a már kezelt kivételt.

Az alábbi egyszerű dinamikus tömb példában, az indexelés műveletfüggvényben kivétel lép fel, ha az index az indexhatárokon kívülre esik – ekkor a program futása véget ér.

```
class DTomb {
protected:
    int meret;
    int * p;
public:
    DTomb(int m = 100);
    int & operator[] (int index);
    ~DTomb() { delete[] p; p=NULL; }
};

DTomb::DTomb(int m) {
    p = new int[meret=m];
}

int & DTomb::operator[] (int index)
{
    if (index<0 || index>=meret) throw "Indexhiba";
    return p[index];
}

void main() {
    try {
        DTomb a(12);
        a[-1] = 23;
    }
    catch(int h) {
        cout<<"Elkapott int kivétel: "<<h<<endl;
    }
    catch(const char * s) {
        cout<<s<<endl;
    }
    catch(...) {
        cout<<"Egyéb kivétel"<<endl;
    }
}
```

Szükség esetén gondoskodhatunk arról, hogy a kivétel csak feldolgozása után jusson ki a `operator[]()` függvényből:

```
int &DTomb::operator[] (int index)
{
    try {
        if (index < 0 || index >= meret)
            throw "Indexhiba"; // Ok!
        return p[index];
    }
    catch (...) {
        cout << "Feldolgoztuk" << endl;
        throw; // Továbbítjuk
    }
}
```

Amennyiben a kivételt le akarjuk „nyelni”, a függvény visszatarozási értékének megadása jelent gondot.

```
int &DTomb::operator[] (int index)
{
    static int szemetes = 0;
    try {
        if (index < 0 || index >= meret)
            throw "Indexhiba"; // Ok!
        return p[index];
    }
    catch (...) {
        cout << "Hibas index: " << index << endl;
        return szemetes;
    }
}
```

Ebben az esetben a programunk tovább fut, és a destruktortól idő előtti aktiválása is elmarad.

Akár egy szinten tekintjük, akár osztály-hierarchián belül, megoldhatatlan gondot jelent a konstruktorban fellépő, illetve a destruktortban kiváltott kivételek kezelése. Amennyiben a konstruktor futása kivétel miatt megszakad, az éppen inicializált objektum hiányosan jön létre, és az elmaradt destruktorthívás következtében a dinamikusan foglalt memóriaterületek is elvesznek.

Mivel nem létezik egyszerű megoldás erre a problémákra, az esetek többségében drasztikusan lépünk fel. Nem engedjük ki a kivételt a konstruktorokból és a destruktorkokból, elfogjuk azt, és üzenet kíséretében befejezzük a program futását:

```

DTomb::DTomb(int m) throw() // a kivételek továbbíttásának
{
    try {
        p = new int[m];
        meret = m;
    }
    catch (...) { // minden kivétel
        cerr<<"Konstrukciós hiba"<<endl;
        exit(-1);
    }
}

```

A lefoglalt memóriaterület automatikus felszabadításáról úgy gondoskodhatunk, hogy a fenti `new` művelettel kijelölt tömböt a szabványos sablonkönyvtár `vector` elemével helyettesítsük. Az így elkészített `vector` objektum törli önmagát, ha a programban valamilyen kivétel továbbítódik, így nincs szükség saját destruktort definiálására.

```

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

class DTomb {
protected:
    vector<int> p;
    int meret;
public:
    DTomb(int m = 100);
    int & operator[] (int index);
    ~DTomb() {}
};

DTomb::DTomb(int m): p(m,0) { meret=m; }

int & DTomb::operator[] (int index) {
    if (index<0 || index>meret) throw "Indexhiba";
    return p[index];
}

void main() {
    try {
        DTomb a(12);
        a[-1] = 23;
    }
    catch(int h) {
        cout<<"Elkapott int kivétel: "<<h<<endl;
    }
    catch(const char * s) {
        cout<<s<<endl;
    }
    catch(...) {
        cout<<"Egyéb kivétel"<<endl;
    }
}

```

Visszatérve az eredeti programváltozathoz, gondoljuk el, mi történik akkor, amikor elindul az indexhibát jelző kivétel, meghívódik a destruktork, és a destruktorkban is kivétel keletkezik. A jelenséget a destruktork ismétlődő hívása kíséri, ezért itt is érdemes elnyelni a kivételt. (A **delete** operátor helyesen értelezi a *NULL* értékű mutatókat.)

```
DTomb::~~DTomb() {
    throw();
}
try {
    delete[] p; p=NULL;
}
catch (...) {
    delete[] p; p=NULL;
}
```

Még nehezebben kezelhető egy osztály-hierarchia valamelyik konstruktorkában, illetve destruktorkában fellépő kivétel. Egy részben inicializált osztálypéldány igazából nem is tekinthető objektumnak, ezért jobban tesszük, ha a programunk azonnal befejezi működését. Talán valami javít a helyzetben, ha a konstruktorkokban és a destruktorkban csak a leggyorsabb, az objektumot kezdőértékkel ellátó műveletekre szorítkozunk. Ugyancsak javít a helyzeten a szabványos adatszerkezetek alkalmazása (STL), amelyek gondoskodnak önmagukat megsemmisítéséről, ha kivétel továbbítódik. Így a saját készítésű destruktork is feleslegessé válhat, és az alapértelmezett destruktorkban nem ke-

letkezik kivétel.

16. Általánosított osztályok (templates)

A legtöbb típusos nyelv megoldásai típusfüggők, vagyis ha elkészítünk egy hasznos függvényt vagy osztályt, az csak a benne rögzített típusú adatokkal működik helyesen. Amennyiben egy másik típusal is szükségünk van a megoldásra, újra meg kell írunk azt, a típusok kicserélésével. Ennek a nem igazán lekezelhető típuscseréigényes programozási módszer kellene lennie. A programozó vállalja a C++ fordító a sablonok bevezetésével. A programozó egyetlen feladatra „megálmodni” a szükségesség függvényi vagy osztályi, bejelölve a lecserélendő típusokat, és a többi már a fordító feladata.

Példaként tekintünk az egydimenziós, 123-elemű egész tömbök egyszerűsített, index-határ ellenőrzése mellett működő típusát!

```
class Intomb {  
public:  
    Intomb(bool nullaz = true) : meret (123) {  
        if (nullaz) memset(tar, 0, 123*sizeof(int));  
    }  
    int & operator [] (int index) ;  
    const int meret;  
private:  
    int tar[123];  
};  
  
int & Intomb::operator [] (int index) {  
    if (index<0 || index>=123) assert(0); // indexhiba  
    return tar[index]; // sikerült  
}
```

Mivel „tud” többet ez a tömb a C++ nyelv statikus tömbjénél? Az adatátírás itt is statikus, és ezt a példa jobb érthetősége érdekében meg is tartjuk. (A fejezet végén egy dinamikus általánosított tömböt is bemutatunk.) A tömb elkészítésekor minden elem alaphehelyzetben lenullázódik,

Intomb a;

kivéve, ha a konstruktort **false** argumentummal hívjuk.

Intomb a(false);

A tömb a *meret* konstans tagban tárolja az elemek számát, illetve átdefiniálja az indexelés operátorát. Ezek alapján a tömb elemeinek elérése:

```
Intomb a;  
a[3] = 12;  
a[7] = 23;  
for (int i=0; i<a.meret; i++)  
    cout<<a[i]<<'\t';
```

Mi tegyünk azonban, ha nem 123 elemet szeretnénk tárolni, vagy ha **double** típusú adatokra van szükségünk. A megoldást az osztálysablon adja, amelyet a fenti *IntTomb* osztály aláhúzott **int** típusainak általánosításával, és az elemszám (123) paraméterként való használatával készíthetünk el.

```
template<class Típus, int elemszam>
class Tomb {
public:
    Tomb(bool nullaz=true): meret(elemszam) {
        if (nullaz) memset(tar, 0, elemszam*sizeof(Típus));
        Típus & operator[] (int index);
        const int meret;
    private:
        Típus tar[elemszam];
};
```

A külső műveletfüggvény esetén az osztály nevét az általánosított típussal és a paraméterrel együtt kell szerepeltetni *Tomb<típus, elemszam>*:

```
template<class Típus, int elemszam>
Típus & Tomb<típus, elemszam>::operator[] (int index) {
    if (index<0 || index>=elemszam) assert(0); // indexhiba
    return tar[index]; // sikerült
}
```

Felhívjuk a függelmezt, hogy az osztálysablon nem implicit **inline** tagfüggvényeit minden olyan forrásállományba be kell építenünk, amelyből azokat hívjuk. E nélkül a forrásállományba a függvény forráskódját előállítani. Több forrásmódulból álló projekt esetén az osztálysablon elemait (a kívüli definiált tagfüggvényekkel együtt) egyetlen deklarációs állományba helyezhetjük, melyet aztán minden forrásmódulba beillesztünk, anélkül hogy „*többszörösen definiált szimbólum*” hibajelzést kapnánk.

Az eddigi munkánk öncélúnak tűnhet, hiszen még nem tudjuk, hogyan használjuk az elkészült osztálysablon (általánosított osztály). Az osztálysablon lényege (a már bemutatott függvényesablonokhoz hasonlóan), hogy a sablon (template) alapján a fordító állítja elő a valóságos, típusfüggő osztályt, annak minden összetevőjével együtt. Kérhetjük egyszerűen a típus elkészítését,

```
typedef Tomb<int, 123> IntTomb;
```

amelyet a szokásos módon használhatunk objektumok előállítására:

```
IntTomb av, bv(false);
```

Azonban közvetlenül az objektumok létrehozására is van mód:

```
Tomb<int, 123> av, bv(false);
```


A sablondefinícióban szereplő *meret* egy konstans paraméter, melynek értékét a fordítás során használja fel a fordító. A sablon feldolgozása során, a paraméter helyén egy konstans értéket, vagy C++ konstans (const) szerepeltethetünk. Természetesen konstans paraméter nélkülű sablonokat is készíthetünk, mint ahogy ezt a fejezet további részeiben tesszük.

Mielőtt azonban tovább mennénk, nézzük meg, mit is nyújt számunkra az új osztály-sablon! A legegyszerűbb alkalmazást már láttunk, így csak a teljesség kedvéért szerepeltetjük újra:

```
Tomb<int, 123> a;
a[3] = 12;
a[7] = 23;
for (int i=0; i<a.meret; i++)
    cout<<a[i]<<'\t';
cout<<<<endl;
```

Az elkészült sablon numerikus adatok mellett sztring-, és osztálytömbök kialakítására is használható. Felhívjuk a figyelmet arra, hogy osztálytípusú tömbelemek esetén (mint a *b2*-ben) az általunk alkalmazott nullázási megoldás túl drasztikus, így ezt kerülünk kell, a sablon konstruktorának *false* értékel való hívásával. Először lefutnak a tömbelem-objektumok konstruktorai, majd pedig a *Tomb<string, 8>* osztály konstruktora következik, ami alaphelyzetben törli a már inicializált elemobjektumok területét.

```
const int ameret=8;
Tomb<char*, ameret> b1;
b1[1] = "Natalia";
b1[7] = "Lafénita";
for (int i=0; i<b1.meret; i++)
    if (b1[i]) cout<<b1[i]<<'\t';
cout<<endl;

Tomb<string, 8> b2 (false);
b2[1] = "Natalia";
b2[7] = "Lafénita";
for (int i=0; i<b2.meret; i++)
    cout<<b2[i]<<'\t';
cout<<endl;
```

Természetesen dinamikusan is létrehozhatjuk a tömbobjektumot, ekkor azonban figyelni kell az indexelés helyes használatára és *meret* adattag elérésre:

```
Tomb<double, 3> *dt;
dt = new Tomb<double, 3>;
(*dt)[0] = 12.23;
dt[0][1] = 34.45;
for (int i=0; i<dt->meret; i++)
    cout<<(*dt)[i]<<'\t';
delete dt; cout<<endl;
```

Mi történik akkor, ha a dinamikus memóriafoglalás során több példányt is létrehozunk? Az eredmény (a példában) *Tomb<double, 3>* típusú elemeket tároló 5-elemű dinamikus tömb. A **double** típusú adatalemek eléréséhez válasszuk a dupla indexelési! Az első index a dinamikus tömbön belül jelöli ki az elemet, míg a második *Tomb<double, 3>* típusú objektumon belül. Ezzel a megoldással – látszólag – egy két-dimenziós, **double** típusú tömböt kapunk.

```
Tomb<double, 3> *dm;
dm = new Tomb<double, 3> [ 5 ];
dm[0][1] = 12.23;
dm[4][2] = 34.45;
for (int i=0; i<5; i++) {
    for (int j=0; j<dm[0].meret; j++)
        cout<<dm[i][j]<<"\t";
    cout<<endl;
}
delete [] dm;
cout<<endl;
```

Amennyiben a dinamikus memóriafoglalás helyett statikus tömböt készítünk, hasonló megoldáshoz jutunk:

```
Tomb<int, 3> m[5];
m[0][1] = 12;
m[4][2] = 23;
for (int i=0; i<5; i++) {
    for (int j=0; j<m[0].meret; j++)
        cout<<m[i][j]<<"\t";
    cout<<endl;
}
cout<<endl;
```

Végezetül tekintsük az alábbi példányosítást, ahol olyan 5-elemű objektumtömböt hozunk létre, melynek mindegyik eleme *Tomb<int,3>* típusú! Az eredmény a programozó számára most is egyfajta két-dimenziós, **int** típusú tömb.

```
Tomb<Tomb<int,3>, 5> p(false);
p[0][1] = 12;
p[4][2] = 23;
for (int i=0; i<p.meret; i++) {
    for (int j=0; j<p[0].meret; j++)
        cout<<p[i][j]<<"\t";
    cout<<endl;
}
cout<<endl;
```

A példából jól látható, hogy a osztálysablon hatékony programozási eszköz, melynek alkalmazása során azonban – sok esetben – a fordító „fejével” kell gondolkodnunk, és ismernünk kell a C++ nyelv teljes, objektum-orientált eszköztárának működését, lehetőségeit. Különösen igaz ez a megállapítás, ha sablonosztályok egymásra épülő rend-

szert (hierarchiáját) kívánjuk megvalósítani. A következőkben rendszerezett, tematikus formában áttekinthetjük a template-készítés és a sablonhasználat során felmerülő fogalmakat és technikákat.

16.1 Általánosított osztály definiálása

A paraméterezett (általánosított) osztály (*generic class*, vagy *class generator*), lehetővé teszi, hogy más osztályok definiálásához a paraméterezett osztályt, mint mintát használjuk. Ezáltal egy adott osztálydefiniáció minden típus esetén alkalmazható.

A sablonosított általános alakja, ahol *tipus1*, ..., *tipusn* az általánosított típusokat jelölik:

```
template <class típus1, ..., class típusn> class osztálynév {
```

```
}
```

Az osztály nem **inline** tagfüggvényeinek definiációját az alábbiak szerint kell megadniunk:

```
template <class típus1, ..., class típusn >
    fvtípus osztálynév< típus1, ..., típusn> :: fvnév(paraméterlista) {
```

```
}
```

Példaként tekintsük a *Pont* osztályból készített általánosított osztályt!

```
template <class típus>
class Pont {
private:
    típus x,y;
public:
    Pont(típus a=0, típus b=0) { setx(a,b); }
    típus getx() { return x; }
    típus gety() { return y; }
    void setx(típus a, típus b) { x=a; y=b; }
};
```

16.2 Specializáció és példányosítás

A sablonokat különböző módon használhatjuk. Az *implicit* példányosítás során (*instantiation*) az általánosított típusokat konkrét típusokkal helyettesítjük. Ekkor először létrejön az osztály adott típusú változata (ha még nem létezett), majd pedig az objektumpéldány:

```
Pont<double> p1, p2(12,23);
```

Explicit példányosítás során arra kérjük a fordítót, hogy hozza létre az osztály példányát a megadott típusok felhasználásával, így az objektum készítésekor már kész osztályal dolgozhat:

```
template class Pont<double>;
Pont<double> p1, p2(12,23);
```

Vannak esetek, amikor a sablon felhasználását könnyíti, ha az általános változatot valamilyen szempont szerint specializáljuk (*explicit specialization*). Az alábbi deklarációnak közül az első az általános sablont, a második a mutatókhoz készített változatot, a harmadik pedig a `void*` mutatókra specializált változatot tartalmazza.

```
template <class típus> class Pont {
    // a fenti osztálysablon
};

template <class típus> class Pont <típus * > {
    // el kell készíteni!
};

template < > class Pont <void * > {
    // el kell készíteni!
};
```

A specializált változatokat az alábbi példányosítások során használhatjuk:

```
Pont<double> pa;
Pont<int > pp;
Pont<void * > pv;
```

A teljeség érdekében függvénysablonok esetén is tekintünk át a példányosítás és a specializáció működését!

```
#include <iostream>
#include <string>
using namespace std;
// általános függvénysablon
template <class típus>
int compare(típus a, típus b) {
    return int(a-b);
}
// explicit példányosítás
template int compare(int, int);
// explicit specializáció char * típusra
template < > int compare<char *>(char *p, char *q) {
    return strcmp(p,q);
}
```

```

void main() {
    int m;
    // az általános függvény hívódik meg
    m = compare(12.7, 23.7); // implicit példányosítás
    char *s1="Lafentita", *s2="Nata";
    // a specializált függvény hívódik meg
    m=compare(s1,s2);
    cin.get();
}

```

16.3 A sablonosztály „barátai” és statikus adatai

Az osztálysablonnak is lehetnek barátai, melyek többféleképpen viselkedhetnek. Azok amelyek nem tartalmazznak sablonelőírást, minden specializált osztály közös barátai lesznek. Ellenkező esetben a külső függény csak az adott példányosított változat *friend* függvényeként használható.

Az általánosított osztályban definiált statikus adattagokat osztálypéldányonként létre kell hoznunk.

16.4 Érték- és alapértelmezett sablonparaméterek

A fejezet bevezető példájában is paramétereztük az osztálysablon, egy egész típusú paraméterrel, melynek a felhasználás során konstans értéket adtunk. Lássuk el a *Tomb* nevű sablon paramétereit alapértelmezés szerinti értékekkel!

```

template<class T, int elemszam=123>
class Tomb {
public:
    Tomb(bool nullaz=true): meret(elemszam) {
        if (nullaz) memset(tar, 0, elemszam*sizeof(T));
        // tipus & operator [] (int index);
        const int meret;
private:
        T tar[elemszam];
    ...
}

```

Ebben az esetben az *int Tomb* típus létrehozásához argumentumok nélkül is speciálizálhatjuk az általánosított osztályunkat:

```
typedef Tomb<> IntTomb;
```

Az alábbi egyszerű példa bemutatja a verem (*stack*) adatstruktúra sablonosztályként történő megvalósítását. A verem sablon paramétereit szintén alapértelmezett értékekkel láttuk el.

```

#include <iostream>
using namespace std;

```

```

template<class T> Tpus=int, int MaxMeret=100>
class Stack {
    Tpus tomb[MaxMeret];
public:
    int sp;
    Stack(void) { sp = 0; };
    void Push(Tpus adat) {
        if (sp<MaxMeret) tomb[sp++] = adat; };
    Tpus Pop(void) { return sp>0? tomb[--sp] : 0; };
    bool Ures(void) { return sp==0; };
};

```

```

void main(void)
{
    Stack<double> dstack; // 1000 elemű double verem
    Stack<char> * sstack; // 100 elemű char * verem
    Stack<> istack; // 100 elemű int verem
    int a=134, b=307;
    istack.Push(a);
    istack.Push(b);
    a=istack.Pop();
    b=istack.Pop();
    sstack.Push("nyelv");
    sstack.Push("C++");
    do {
        cout << sstack.Pop()<<endl;
    } while (!istack.Ures());
}

```

16.5 A typename kulcsszó

A **typename** kulcsszó a típussablonok használatához köti. Definíciójában az osztályok esetén a **class** kulcsszó helyett használhatjuk a típusdefiniációban (**typedef**), illetve a sablondeklarációkban is szerepeltethetjük a **class** kulcsszó helyett.

Az első példában a **typename** kulcsszót azért használjuk, mert a változókat úgy kell definiálnunk, hogy a típus (**T::Tpus**) még nem jött létre:

```

template<class T>
void fV() {
    typedef typename T::Tpus TTpus;
    TTpus a;
    typename T::Tpus b;
    TTpus * pta;
}

class Valami {
public:
    typedef char * Tpus;
};

// Például a függvényhívás: fV<Valami>();

```

A `typename` kulcsszót alkalmazhatjuk a `template` deklarációkban is a `class` helyett:

```
template <typename T1p1, typename T1p2>
T1p2 konverzio (T1p1 t1) {
    return (T1p2)t1;
}

template <typename T1px, class T1py>
bool Egyenlo (T1px x, T1py y) {
    return x==y;
}

void main() {
    bool x=Egyenlo(10,20);
    double xc=Konverzio<double,int>(10);
}
```

16.6 Összetettebb sablonpelda

Sablonhierarchia (sablonosztályok hierarchiájának) kialakításához nagy gyakorlatra van szükség, ezért csak egy szintű sablonosztályok készítésével foglalkozunk.

Készítsük el a 13. és 14. fejezetekben használt *Vektor* osztály általánosított, kibővített változatát! Egészítsük ki az osztályt egy iterátorral is!

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

template <class T>
class Vektor {
public:
    typedef T* lterator;

    explicit Vektor(int n=10);
    Vektor(const Vektor& v);
    Vektor(T* a, int n);
    ~Vektor() {delete[] p;}
    int fh() const { return meret; }

    T& operator[] (int i);
    Vektor& operator= (const Vektor& v);
    Vektor operator+= (const Vektor& v);
    lterator begin() { return p; }
    lterator end() { return p+meret; }
private:
    T* p;
    int meret;
};
```

Egy lehetséges főprogram, amelyben elvégezzük az osztálysablon példányosítását, vagyis konkrét típus behelyettesítésével való megvalósítását:

```
// Konstruktorok
template<class T> Vektor<T>::Vektor(int n) {
    p = new T[meret=n];
}

template<class T> Vektor<T>::Vektor(const Vektor& v) {
    p = new T[meret=v.meret];
    for (int i = 0; i < meret; ++i)
        p[i] = v.p[i];
}

template<class T> Vektor<T>::Vektor(const T a[], int n) {
    p = new T[meret=n];
    for (int i = 0; i < meret; ++i)
        p[i] = a[i];
}

// osztálytag műveletfüggvények
template<class T> T& Vektor<T>::operator[](int i) {
    if (i < 0 || i > meret-1) throw i;
    return p[i];
}

template<class T> Vektor<T>& Vektor<T>::operator=
(const Vektor<T>& v) {
    delete []p;
    p=new T[meret=v.meret];
    for (int i = 0; i < meret; ++i)
        p[i] = v.p[i];
    return *this;
}

template<class T> Vektor<T> Vektor<T>::operator+
(const Vektor<T>& v) {
    int m = (meret > v.meret) ? meret : v.meret;
    for (int i = 0; i < m; ++i)
        p[i] += v.p[i];
    return *this;
}

// külső műveletfüggvény
template<class T> T& Vektor<T>::operator+
(const T& v1, const T& v2) {
    T& sum = v1;
    sum+=v2;
    return sum;
}
```



```

void main() {
    int a[5]={1,2,3,4,5};
    int b[3]={7,12,23};

    // Példányosítás typedef-ben
    typedef Vektor<int> iVektor;
    iVektor v1(a,5), v2(b,3), v3;
    v3=v1;
    v3=v2+v1;

    // Lokális példányosítás double típussal
    Vektor<double>::iVektor p;
    Vektor<double> x,y,z;
    for (p=x.begin(); p!=x.end(); p++)
        *p=1223.79;
    y=x;
    z=x+y;
}

```


A C++ nyelv könyvtára

Függelék

Fl. A szabványos C++ nyelv könyvtárainak áttekintése

A szabványos C++ különböző összetevőkből épül fel. Egyrészt átvette a szabványos (ANSI) C nyelv függvénykönyvtárát, amely egy sor jól használható függvényt és típust tartalmaz. Az átvett könyvtárak lehetőségeit (például *limits*) néhány esetben C++ átvárban is megtaláljuk (például *limits*). A C++ nyelvhez kapcsolódóan új elemek is megjelentek (például *new*, *exception*, *istream* stb.). A C++ könyvtár elemeit az alábbiak szerint csoportosíthatjuk. A csoportosítás során kiemeltük a szabványos sablon-könyvtárhoz (*Standard Template Library* – *STL*) tartozó állományokat.

<i>A C++ nyelvvel támogató könyvtár</i>	
Típusok (<i>NULL</i> , <i>size_t</i> stb.)	<stddef>
Az implementáció jellemzői	<limits> <limits> <float>
Programindítás és -befejezés	<csdlib>
Dinamikus memóriakezelés	<new>
Típusazonosítás	<typeinfo>
Kivételkezelés	<exception>
Egyéb futásidejű támogatás	<csdarg> <csjtmp> <ctime> <csignal> <csdlib>
<i>Hibakezelési könyvtár</i>	
Kivételoszállyok	<stdexcept>
Assert makrók	<cassert>
Hibakódok	<cerrno>
<i>Általános szolgáltatások könyvtára</i>	
Műveleti elemek (<i>STL</i>)	<utility>
Műveletobjektumok (<i>STL</i>)	<functional>
Memóriakezelés (<i>STL</i>)	<memory>
Dátum- és időkezelés	<ctime>

String könyvtár

String oszállyok

Nullavégű karakterláncok kezelését segítő függvények

<string>
<cctype>
<cwctype>
<cstring>
<wchar>
<csdlib>

Országfüggetlő (helyi) beállítások könyvtára

A helyi sajátosságok szabványos kezelés

A C könyvtár helyi beállításai

<locale>
<locale>

A tárolók könyvtára (STL)

Adatsorok (STL)

<deque>
<list>
<queue>
<stack>
<vector>
<map>
<set>
<bitset>

Iterátorok (általánosított mutatók) könyvtára (STL)

Iterátorelemek, előre definiált iterátorok, adatfolyam-iterátorok (STL)

<iterator>

Algoritmusok könyvtára

Adatsorkezelés, rendezés, keresés stb. (STL)

<algorithm>
<csdlib>

Numerikus könyvtár

Komplex számok

<complex>

Számítóművek

<valarray>

Általánosított numerikus műveletek (STL)

<numeric>

A C könyvtár numerikus elemei

<cmath>
<csdlib>

Input/output könyvtár

<i>Forward</i> (előrevetett) deklarációk	<iosfwd>
Szabványos <i>iostream</i> objektumok	<iostream>
Az <i>iostream</i> osztályok alaposztálya	<iostream>
Adatfolyam pufferek	<streambuf>
Adatformázás és manipulátorok	<iostream> <ostream> <iomanip>
Sztringadatfolyamok	<sstream> <csdlib>
Fájladatfolyamok	<fstream> <csdio> <wchar>

A szabványos C++ könyvtár azonosítói az *std* névtérületen keresztül érhetjük el. A C++ szabvány a fentiekben kívül tartalmaz még 18 deklarációs állományt a szabványos C-könyvtárból, melyek különbözhetnek az új változataiktól. (A C-könyvtárbeli nevek a globális névtérben tárolódnak)

C deklarációs állomány C++ megfelelő

<ctype.h>	<cctype>
<errno.h>	<cerrno>
<float.h>	<cfloat>
<iso646.h>	<ciso646>
<limits.h>	<climits>
<locale.h>	<clocale>
<math.h>	<cmath>
<setjmp.h>	<csetjmp>
<signal.h>	<csignal>
<stdarg.h>	<csdarg>
<stddef.h>	<csddef>
<stdio.h>	<csdio>
<stdlib.h>	<csdlib>
<string.h>	<cstring>
<time.h>	<ctime>
<wchar.h>	<cwchar>
<wctype.h>	<cwctype>

F2. Adatok bevitele és kivitele

A C++ nyelv nem rendelkezik adatbeviteli és adatkiviteli utasításokkal, minden ilyen feladatot könyvtári függvényekkel, illetve objektumokkal kell megoldanunk.

Az operációs rendszer szintjén a C++ program számára az adatátvitel mindig adattároló-
kimenete pedig a képernyő vagy a nyomtató. Az adatátvitel egylege a bájti (szó), amit
a C++ nyelvben a **char** (**wchar_t**) típus valósít meg.

Az adatfolyam-kezelő (*stream*) függvények és objektumok az állományokat, mint ka-
rakterek folyamát tekintik. A C nyelvtől örökölt lehetőségekhez a *stdio*, míg az
objektum-orientált megoldásokhoz az *iostream* fejlálmányt kell a programunkba be-
építeni.

A legtöbb operációs rendszer a C++ programok indításakor három előre definiált adat-
folyamot automatikusan megnyit, lehetővé téve ezzel az alapvető perifériális egységek
elérését. A C nyelvtől örökölt módon a (*FILE* * típusú) *stdin* a szabványos input-, az
stdout a szabványos output-, míg az *stderr* a szabványos hiba-adatfolyamot jelöli. A
C++ könyvtár objektumokat biztosít ezen adatfolyamok elérésére, melyek rendre a
cin, a *cout* és a *cerr*. (A *wchar_t* típusú széles-karakterekből álló adatfolyamok objekt-
umai a *wcin*, a *wcout* és a *wcerr*). Az elmondottakat táblázatban foglaltuk össze:

C adatfolyam			C++ objektum	Periféria	A művelet iránya
<i>stdin</i>	<i>cin</i>	bíllentyűzet (átírányítható)	<i>input</i>		
<i>stdout</i>	<i>cout</i>	képernyő (átírányítható)	<i>output</i>		
<i>stderr</i>	<i>cerr</i>	képernyő	<i>output</i>		
-	<i>clog</i>	képernyő – a <i>cerr</i> puffereit változata	<i>output</i>		

A következő két alfejezetben először áttekintjük a fenti adatfolyamok elérését támoga-
tó megoldásokat, majd a fejezetet az adattálmányok kezeléséhez szükséges ismeretek
összegeése zárja.

F2.1 A C-könyvtár alapvető I/O műveletei (*stdio*)

A következő táblázatban összefoglaltuk az azonos jellegű input-, illetve outputművele-
tek végzésére alkalmas szabványos C-függvényeket, külön kiemelve a függvények
széleskaraktereket használó párját:

amely egy karaktert olvas a szabványos inputeszközről (billeentyűzetről). Ha az olvasás sikeres volt, akkor a beolvasott karakter kódját kapjuk meg visszatérési értékként. A fájlvége esemény bekövetkeztekor *EOF* értékkel tér vissza a *getchar()*. (Billeentyűzet esetén a $<Ctrl+Z>$ billeentyűkombináció váltja ki ezt az eseményt.)

```
int getch(char (void) ;
```

A legegyszerűbb adatbeviteli függvényyszerű makró a *getchar()*.

```
#define getch()  getc(stdin)
#define putch(c) putc((c) , stdout)
```

Az adatfolyam jellegű adatkezelésnél alapvető műveletnek számít egyetlen karakter beolvasása és kírása, mely feladatok elvégzésére makrókat tartalmaz a szabvány:

F2.1.1 Karakterek kírása és beolvasása

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

A fejezetben bemutatásra kerülő függvények prototípusát a *csdio* fejláblomány tartal-

mazza:

<i>stdin</i>	<i>stdout</i>	<i>string</i>	<i>stream I/O</i>	szeleskarakteres függvények
<i>getchar()</i>	-	-	<i>getc()</i>	<i>getwchar()</i> , <i>getwc()</i>
-	-	-	<i>ungetc()</i>	<i>ungetwc()</i>
<i>fgetchar()</i>	-	-	<i>fgetc()</i>	<i>fgetwchar()</i> , <i>fgetwc()</i>
<i>gets()</i>	-	-	<i>fgets()</i>	<i>getws()</i> , <i>fgetws()</i>
-	-	-	<i>getw()</i>	
<i>putchar()</i>	-	-	<i>putc()</i>	<i>putwchar()</i> , <i>putwc()</i>
<i>fputchar()</i>	-	-	<i>fputc()</i>	<i>fputwchar()</i> , <i>fputwc()</i>
<i>puts()</i>	-	-	<i>fputs()</i>	<i>puts()</i> , <i>fputs()</i>
-	-	-	<i>putw()</i>	<i>putws()</i> , <i>fputws()</i>
-	-	-	<i>fread()</i>	
-	-	-	<i>fwrite()</i>	
<i>scanf()</i>	-	<i>sscanf()</i>	<i>fscanf()</i>	<i>wscanf()</i> , <i>swscanf()</i> , <i>fscanf()</i>
<i>vscanf()</i>	-	<i>vsscanf()</i>	<i>vfscanf()</i>	<i>wvscanf()</i> , <i>swvscanf()</i> , <i>vfscanf()</i>
<i>printf()</i>	<i>sprintf()</i>	<i>fprintf()</i>	<i>fprintf()</i>	<i>swprintf()</i> , <i>swprintf()</i> , <i>fswprintf()</i>
<i>vprintf()</i>	<i>vsprintf()</i>	<i>vfprintf()</i>	<i>vfprintf()</i>	<i>swprintf()</i> , <i>swprintf()</i> , <i>fswprintf()</i>

A legegyszerűbb adatkivitelű függvényyszerű makró a *putcchar()*,

```
int putcchar(int ch);
```

amely a megadott *ch* karaktert kiírja a szabványos outputeszközzre (képernyőre). Sikeres működés esetén a *ch* karakter kódját kapjuk meg visszatérési értékként.

F2.1.2 Karakter sorozatok kiírása és beolvasása

A C-könyvtárban találunk függvényeket a karakter sorozatok (stringek) egyetlen hívással történő beolvasására (*gets()*) és kiírására (*puts()*).

```
char * gets(char * sptr);
```

A *gets()* függvény egy sort (<Enter> lenyomásáig) olvas a szabványos inputról, majd a karaktereket az argumentumban megadott *sptr* mutató által kijelölt területre másolja. A beolvasott stringben az új sor ('*\n*') karakter '*\0*' (EOS) karakterrel helyettesítődik, ezért a függvény nem alkalmas az új sor karakter beolvasására. (Ezre a célra a *getchar()* makró kell használnunk.) A *gets()* visszatérési értéke sikeres olvasás esetén a megadott puffert címe, illetve *NULL*, ha hiba lép fel.

```
int puts(char * sptr);
```

A *puts()* függvény az argumentumban megadott karakter sorozatot a szabványos kimenetre (a képernyőre) írja. A string megjelenítését automatikusan a '*\n*' (új sor) karakter kiírása követi. A függvény egy nem negatív szám visszaadásával jelzi a string sikeres kiírását, ellenkező esetben pedig *EOF* (-1) értékkel tér vissza.

F2.1.3 Formázott adatok bevittele és megjelenítése

A szabványos C-könyvtár *printf()* és *scanf()* függvényeivel megadott formátum szerint lehet kiírni, illetve beolvasni alapítványi adatokat és karakter sorozatokat.

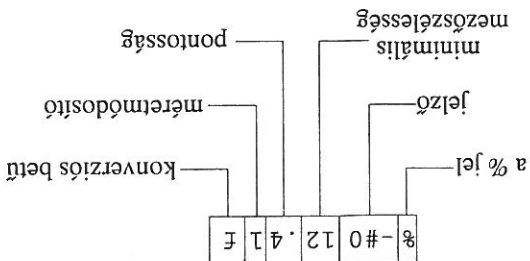
```
A printf() függvény
```

A *printf()* függvény hívásának formája:

```
printf(formátum, argumentumlista);
```

A formátum, amely általában stringkonstans, kétféle karaktert tartalmazhat. Az első csoport karakterei normál karakter sorozatokat alkotnak a formátumsztringen belül, amelyek minden átalakítás nélkül jelennek meg.

A másik csoport karakterei speciális műveleteket írnak elő, amelyek egyrészt meghatározzák az argumentumok értelmezésének módját, másrészt pedig a megjelenítés formáját. A konverziós előírások százalékjellel (%) kezdődnek és valamilyen konverziós karakterrel zárulnak. Az előírás teljes formájában hat jól elkülöníthető részre osztható:



Az alábbiakban áttekinthetjük az egyes részek szerepét és a részek megadásának lehetőségeit:

Jelző

- a mínuszjel megadása esetén a konvertált argumentum a mező bal széléhez igazítva jelenik meg,

+ (pluszjel) számok kitérésakor az előjel (+ vagy -) mindig megjelenik,

szökő az előjel helyén a mínuszjel vagy szökő jelenik meg,

0 a szökő helyett 0-val töltődik fel a kiírt adat előtti terület,

(hashmark) az ún. alternatív kitérés kép jelle. (A %*xx* formátum hatására a hexadecimális számok előtt a 0x előtag, míg a %*no* formátum esetén az okta-

lis számok előtt a 0 előtag jelenik meg.)

Ha a jelző karaktereket elhagyjuk, akkor az output a megadott mezőben jobbra igazítva jelenik meg, és balról szökő karakterek töltik ki a nem használt területet.

Minimális mezőszélesség

A minimális mezőszélesség decimális jegyeket tartalmazó egész konstans, amely a konvertált argumentum kitérésakor felhasználható mező szélességét határozza meg. Amennyiben az első jegy 0, akkor azt jelzőként értelmezi a rendszer. Ha a megadott mezőben nem fér el az eredmény, akkor az a szükséges méretű mezőben fog megjelenni. Ha a csillag (*) karakterrel adjuk meg a mezőszélességet, akkor a tényleges mezőszélesség-értéket az argumentumlista soron következő eleme adja meg. Az elmondottak alapján az alábbi két kitérés művelet eredménye ugyanaz lesz:

```
printf("%7d", 1223);  
int w=7;  
printf("%*d", w, 1223);
```

Konverziós karakterek

A formátumilem mindig tartalmaz legalább egy betűt, amely alapján az argumentum feldolgozását és konvertálását végzi a rendszer. Minden konverziós karakternek megfelel valamilyen argumentumtípus és nyomtatási kép.

Konverziós karakter	Az argumentum típusa	Nyomtatási kép
c	int	Az unsigned char típusra való konvertálás eredményének megfelelő karakter.
d, i	int	Előjeles decimális egész.
u	unsigned int	Előjele nélküli decimális egész.
f	double	Előjele nélküli decimális egész.
e, E	double	Előjele nélküli decimális egész. A pontoság alapértelmezésben 6, a 0 pontossággal a tizedespont kitírása elnyomható.
g, G	double	Előjele nélküli decimális egész. A pontoság alapértelmezésben 6, a 0 pontossággal a tizedespont kitírása elnyomható.
s	char*	A karaktertípusokat kiírja az EOS -ig, vagy a karaktertípusok határáig.
%	-	Nincs konverzió - kiírja a % karaktert.

Pontososság p

A formátumban a pont után megadott decimális szám a megjelenítés pontososságát határozza meg. A pontososság értelmezése típusonként eltérő:

- egész számok esetén a jegyek minimális számát (*d, i, o, u, x, X*),
- lebegőpontos számok esetén tizedes jegyek (*e, E, f, F*), illetve az értékes jegyek (*g, G*) számát,
- sztring esetén pedig a kiírni kívánt karakterek maximális számát határozza meg.

	%?	1223	1223	2307	4C7	4C7	?	=	X
	%2?	1223	1223	2307	4C7	4C7			
	%6?	1223	1223	2307	4C7	4C7			
	%-6?	1223	1223	2307	4C7	4C7			
	%.7?	0001223	0002307	0004C7	0004C7	0004C7			
	%#?	1223	02307	0X4C7	0X4C7	0X4C7			

Az egész számok decimális, oktális és hexadecimális formátumban történő kiírását a következő táblázat tartalmazza.

%s	Lafenta
%7s	Lafenta
%.7s	Lafent
%23s	Lafenta
%-23s	Lafenta
%23.7s	Lafent
%-23.7s	Lafent

Nezzünk néhány példát a különböző formátum-megadási módokra! Az alábbi táblázatban a "Lafentia" karaktertorsorozat kiírásának eredményét foglaltuk össze. A kiírás-hoz használt formátum-előírást a bal oldali oszlop tartalmazza, míg a nyomtatási kép a jobb oldali oszlopban tanulmányozható (a | karakterek a mezőhatárt jelölik):

<i>H</i>	<i>a d, i, o, u, x</i> és <i>X</i> konverziós karakterrel együtt megadva a short int típust jelöli,
<i>l</i>	<i>a d, i, o, u, x</i> és <i>X</i> konverziós karakterrel együtt megadva a long int típust jelöli,
<i>L</i>	mg az <i>e, E, f, g</i> és <i>G</i> betűk esetén a double típust, az <i>e, E, f, g</i> és <i>G</i> konverziós betűkkel együtt használva a long double típuselöltrés- nak felel meg, mg a <i>d, i, o, u, x</i> és <i>X</i> konverziós karakterrel együtt megadva a long int (bizonyos változatokban a 64-bites — int64 egész) típust jelöli.

A mérőmódszótó betűi közvetlenül a konverziós karakter előtt áll. Lehetőséges értékei:

Méretimódosító elöttag

me tartalmazza.

Ha a pontosság a specifikációból kimarad, akkor az alapértelmezés szerinti érték (6) érvényesül. Ha a pontosságnak 0-át adunk meg, akkor a d , i , o , u , x , illetve X konverziós karakterek esetén az alapértelmezés szerinti számú jegy kerül kiírásra, az e , E és f konverziós karakterek esetén pedig elmarad a tízdeszpont. A csillag (*) megadása azt jelenti, hogy a tényleges pontossági értéket az argumentumlista soron következő ele-

A formátumban a kérdőjel (?) helyén a táblázat felső sorában megadott konverziós be-
tűket kell behelyettesíteni (a | karakterek a mezőhatárt jelölik):

Végezetül nézzünk néhány példát **double** típusú lebegőpontos értékek konverziójára!

	7.12	-23.0	0.0000023
%f	7.120000	-23.000000	0.000002
%e	7.120000e+00	-2.300000e+01	2.300000e-06
%g	7.12	-23	2.3e-06
%10.11f	7.12	-23.0	0.0
%+5.0E	+7E+00	-2E+01	+2E-06
%+5.1E	+7.1E+00	-2.3E+01	+2.3E-06
%+7.2E	+7.12E+00	-2.30E+01	+2.30E-06

A *scanf()* függvény

Az

```
int scanf(const char *formatum, ...);
```

deklarációjú függvény formázott adatbevittelt valósít meg. A *scanf()* függvény a szab-
ványos bevitteli eszközről olvas adatokat, és az esetek többségében a *printf()* konver-
ziók fordítottját hajtja végre. Hívásának általános alakja:

```
scanf(formatum, argumentumlista);
```

A *scanf()* függvény karaktereket olvas a szabványos adatbevitteli eszközről, majd a
formátumsztring specifikációi szerint értelmezi, és konvertálja azokat. A művelet
eredményét a soron következő argumentum által kijelölt memóriaterületen tárolja.

A *scanf()* befejezi az olvasást, ha a specifikációnak megfelelő számú adatot már fel-
dolgozott, vagy ha az adatbevittelt valamely oknál fogva nem felel meg a formátum-
sztring előírásainak. A *scanf()* visszatérési értéke a sikeresen beolvasott adatok száma,
ami jól használható arra, hogy ellenőrizzük, hogy valóban annyi adatot olvasott-e be a
programunk, mint ahányat előírtunk. A 0 visszatérési érték azt jelöli, hogy egyetlen
változónk sem kapott értéket a *scanf()* hívása során.

A formátumsztring felépítése hasonlít a *printf()* függvénynél használt formátumra,
azonban több ponton is különbözik attól. A *scanf()* formátumsztring az alábbi elemek-
ből épül fel:

- tagoló (*whitespace*) karakterek, melyeket a *scanf()* figyelmen kívül hagy,
- konverziós előírások a % karakterrel kezdődnek, és opcionálisan tartalmazhat-
ják a:

*	csillag karaktert (melynek hatására a beolvasott adatot a <i>scanf()</i> eldobja), mészélességet megadó decimális egész számot,
<i>n</i>	opcionális méretkijelölő karaktert: <i>h</i> (short), <i>l</i> (long vagy double), <i>L</i> (long double).

A specifikációt a konverziós karakter zárja, amelynek lehetséges értékeit az alábbi táblázatban foglaltuk össze.

Konverziós karakter	Bevit adat	Argumentum típusa
d D , (ld)	Decimális egész.	int *
i I , (li)	Egész szám, akár oktális (0), akár hexadecimális (0x vagy 0X) formában.	long *
o O , (lo)	Oktális egész a 0 előtag nélkül.	long *
u U , (lu)	Előjel nélküli decimális egész.	unsigned *
x X , (lx)	Hexadecimális egész, akár a 0x vagy 0X előtaggal, akár anélkül.	unsigned long *
C	Karakter(ek). A soron következő karakter (alapértelmezésben csak egy) kerül a megfelelő memóriaterületre. A tagoló karakter is beolvasásra kerülnek. A következő nem tagoló karakter olvasásához a %Is specifikációt használjuk.	char *
S	Sztring olvasása tagoló karakterig. A specifikációhoz tartozó pointer egy olyan tömbre kell mutatson, amely elegendően nagy a beolvasandó karakterek és a sztring végét jelző EOS tárolására.	float *
f , e , g , E , G	E három vezérlőkarakter szolgál lebegőpontos számok beolvasására. Az előjel, a tizedespoint és az exponens megadása opcionális. Ha a specifikációban az l előtag is szerepel (például %lf), akkor az argumentum típusa: Az L előtag esetén az argumentum típusa pedig: Az L előtag esetén az argumentum típusa pedig:	double *
P	A printf által kirt formatumú mutató értékét olvassa be.	long double *
n	A formatumig beolvasott karakterek számát adja vissza a függvény az argumentummal kijelölt egész (int) típusú változóban.	int *

Ha az adatbevitel során a százalékjellet kell megadnunk, akkor a formátumsztringben a %% elemet kell szerepeltetni. Az alábbiakban a c és az n konverziós karakterek együttes alkalmazására mutatunk példát. A program 10 tetszőleges karaktert tartalmazó karaktertörzsort olvas be:

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

void main() {
    int a;
    char s[20];
    scanf("%10c%n", s, &a); // az a értéke mindig 10 lesz
    cout<<s<<endl;
    cin.sync_with_stdio(false); // nincs szinkronizálás
    cin.get();
}
```

A *scanf()* szűrni is képes a beolvasandó karaktertörzsort karaktereit. Ha a százalékjellet után álló szögletes zárójelék között karakterterhalmazi adunk meg, akkor csak az ott szereplő karakterek kerülnek be a sztringbe (szűrés).

```
char str[80];
scanf("%[I\Nin]", str);
```

csak a felsorolt karaktereket tartalmazó karaktertörzsortot olvassa be az input-sorból (a példa csak kis- és nagy I, N betűket tartalmazó szöveget olvassa be);

```
scanf("%[0-9a-fA-F]", str);
```

csak a megadott intervallumba eső karaktereket olvassa be (a példa csak hexa-decimális jegyeket tartalmazó sztringet fogad el);

```
scanf("%[^abcd]", str);
```

csak azokat a karaktereket olvassa be, amelyek nem szerepelnek a felsorolás-ban;

```
scanf("%[^\0-9]", str);
```

csak a megadott karakter-intervallumon kívülre eső karaktereket olvassa be (a példa minden számjegyet nem tartalmazó sztringet elfogad). Természetesen a karakterterhalmaz kijelölésénél a fenti megoldásokat együttesen is alkalmazhatjuk.

F2.1.4 Írás stringbe és olvasás stringből

C++ nyelven adott a lehetőség olyan formázott adatátviteli műveletek végzésére, amelyek karaktertípusokból olvasnak. A stringbe való for-mázott kitérő az *sprintf()* függvényt használjuk, melynek prototípusa:

```
int sprintf(char* puffer, const char* formatum, ...);
```

A stringből való olvasás művelete az *scanf()* függvényel végezhető el:

```
int scanf(const char* puffer, const char* formatum, ...);
```

A fenti két függvény jól alkalmazható különböző típusú számok és szövegek közötti konverzió elvégzésére.

A függvények használatát az alábbi program szemlélteti, ahol a stringtömbből olvasunk a **double** értékeket, amiket kerekített **long** típusú értékként írunk vissza a stringtömbbe:

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

void main()
{
    char adat[3][16] = {"22.7", "12.23", "1979.12"};
    double e;
    int i;

    for (i=0; i<3; i++)
    {
        sscanf(adat[i], "%lf", &e);
        e += 0.5; // kerekítés
        sprintf(adat[i], "%04ld", long(e));
    }

    for (i=0; i<3; i++)
        printf("%d. \t %s\n", i, adat[i]);

    cin.sync_with_stdio(false);
    cin.get();
}
```

0.	0023
1.	0012
2.	1979

F2.1.5 Az stdio és stdout adatfolyamok átírányítása

A legtöbb operációs rendszer (a Unix-hoz hasonlóan) támogatja a szabványos be- és kimenetek átírányítását. A C++ programunkon belül semmit sem kell tennünk az átírányítás megvalósítása érdekében. Tekintsük a *CHARIO.CPP* példaprogramot, melynek készítsük el a futtatható változatát (*CHARIO.EXE*)!

```
// CHARIO.CPP
#include <cstdio>
using namespace std;

void main() {
    int c;
    while ((c = getchar()) != EOF)
        putchar(c);
}
```

Ezek után parancssor készületi jelenél futtatva a *CHARIO.EXE* programot, használjuk az operációs rendszer adatátírányító operátorait (<, >, >>);

CHARIO
Indítás után a begépet karaktert az <Enter> lenyomása után visszairja a képernyőre a program. A kilépéshez a <Ctrl+Z> billentyűt kell megnyomunk.

CHARIO > ADAT.TXT
A begépet karakterek az *ADAT.TXT* szöveges állományba másolódna. A kilépéshez a <Ctrl+Z> billentyűt kell megnyomunk. (Az *stdout* átírányítása az *adat.txt* állományba.)

CHARIO > ADAT.TXT
A program megjeleníti a képernyőn az *ADAT.TXT* szövegfájl tartalmát. (Az *stdin* átírányítása az *ADAT.TXT* állományból.)

CHARIO > ADAT.TXT > ADAT2.TXT
A program az *ADAT.TXT* szöveges állomány tartalmát átírja az *adat2.txt* fájlba. (Az *stdin* az *ADAT.TXT* állományt, míg az *stdout* *ADAT2.TXT* állományt jelenti.)

Az átírányítás segítségével interaktív programot is lehet futtatni állományban tárolt adatokkal, illetve a program kimenetét fájlba vagy nyomtatóra lehet írányítani.

F2.2 A C++ alapvető I/O műveletei (iostream)

Mint a fejezet bevezetőjében említettük, a C++ nyelv objektumokkal támogatja az input/output műveletek végzését. Az objektumok deklarációját az *iostream* fejelet állomány tartalmazza:

<i>deklaráció</i>	<i>leírás</i>
extern ostream cout;	szabványos adatkivitel (átírányítható),
extern istream cin;	szabványos adatbevitel (átírányítható),
extern ostream cerr;	szabványos hiba-adatfolyam,
extern ostream clog;	szabványos puffert használó hiba-adatfolyam.

Az objektumok eléréséhez az alábbi megoldások közül választhatunk:

<i>a deklarációk elérése</i>	<i>felhasználás</i>
<pre>#include <iostream> using namespace std; void main() { int a; cin>>a; cin.get(); cout<<a<<endl; cin.get(); }</pre>	<pre>void main() { int a; std::cin>>a; std::cin.get(); std::cout<<a<< std::endl; std::cin.get(); }</pre>

Karakteres (**char**) adatfolyamokon végezhető műveletekhez a biteitlő operátorok át-definiált változatai, illetve az objektumok tagfüggvényei közül egyaránt választhatunk.

F2.2.1 Az << és a >> műveletek

Az esetek többségében az adatbevitelt a **cout** objektum << operátorának, míg az adatkivítelt a **cin** objektum >> operátorának tültérhelt változataival végzzük. Az alábbi táblázatban összefoglaltuk, hogy mely C++ típusok ismeretére készítették fel az objektumok oszlályait:

bool	char, signed char, unsigned char
char	short, unsigned short
int	int, unsigned int
long	long, unsigned long
void *	unsigned char *
float	double
	long double

Felhívjuk a figyelmet arra, hogy karaktertípusok beolvasása esetén az adatbevitelt az első elválasztó karaktermel (például szóköz) megszakad. Ezért írásjelekkel tagolt szövegek bevitelére tagfüggvényt kell használnunk, amely a bekért szöveg hosszát is ellenőrzi, illetve adott karakterig olvassa be a szöveget.

Nézzünk néhány megoldást a tagfüggvények alkalmazására!

Egyetlen karakter beolvasása:

```
int ch=cin.get(); // vagy
cin.get(ch);
```

Egyetlen karakter kiírása:

```
cout.put('N');
```

Legfeljebb 12-karakteres szöveg beolvasása

```
char s[13];
cin.get(s,13,'');
```

Legfeljebb 23-karakteres szöveg beolvasása

```
char n[24];
cin.getline(n,24);
```

Felhívjuk a figyelmet arra, hogy az inputpuffer fel nem dolgozott karakterei a következő művelet rendelkezésére állnak. A felesleges adatoktól például az alábbi hívással szabadulhatunk meg:

```
cin.ignore(79, '\n');
```

Fr.2.2 I/O manipulátorok használata

A *cin* és a *cout* objektumok egy sor beállítási lehetőséggel rendelkeznek, melynek segítségével szabályozhatjuk a műveletek elvégzését. Az ún. formátumjelző bitek beállításához, illetve törléséhez az *ios_base* alapszály *enum* konstansait, illetve tagfüggvényeit használjuk.

```
inline long ios::flags() const;
inline long ios::flags(long _l);
inline long ios::setf(long _f, long _m);
inline long ios::setf(long _l);
inline long ios::unsetf(long _l);
```

Mivel a bitek beállításaánál, illetve törlésénél egyszerűbb megoldás is rendelkezésünkre áll ún. manipulátorok formájában, így a tárgyalásnál azokra helyezzük a hangsúlyt. A manipulátorok olyan speciális függvények, melyek hívását a <<, illetve a >> operátorok operandusaként adjuk meg. Például, a 2005 hexadecimalis formában I2 pozíción balra igazítva, az alábbi utasítással jeleníthető meg:

```
cout<<left<<hex<<setw(12)<<2005<<endl;
```

A paraméter nélküli manipulátorok deklarációját az *iostream*, míg a paraméterrel rendelkező manipulátorok leírását az *iomanip* fejlécállományok tartalmazzák. A fenti példa működéséhez mindkét deklarációs állományra szükségünk van:

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

Az alábbiakban áttekintjük az alkalmazható manipulátorokat. Érdekeség kedvéért – főleg haladó programozók részére – tagfüggvényekkel elvégezhető alternatív megoldást is mutatunk az egyes manipulátorok helyettesítésére. (Ezek többségében a formátumjelző biteket állítjuk.)

Paraméter nélküli manipulátorok

Az alábbi manipulátorokat párban használjuk, hatásuk a bekapcsolástól a kikapcsolásig terjed:

<i>Manipulátor</i>	<i>Megoldás tagfüggvényekkel</i>	<i>Írány</i>	<i>Leírás (bekapcsolva)</i>
<i>boolalpha</i>	<code>ios.setf(ios_base:: boolalpha) ;</code> <code>ios.unsetf(ios_base:: boolalpha) ;</code>	I/O	A logikai értékek megadhatók a <i>true</i> és a <i>false</i> szavakkal.
<i>showbase</i>	<code>ios.setf(ios_base:: showbase) ;</code> <code>ios.unsetf(ios_base:: showbase) ;</code>	O	A számok előtt a számrendszer jelle (0, vagy 0x) is megjelenik.
<i>showpoint</i>	<code>ios.setf(ios_base:: showpoint) ;</code> <code>ios.unsetf(ios_base:: showpoint) ;</code>	O	A tizedespont mindig megjelenik.
<i>showpos</i>	<code>ios.setf(ios_base:: showpos) ;</code> <code>ios.unsetf(ios_base:: showpos) ;</code>	O	A pozitív előjel is megjelenik.
<i>skipws</i>	<code>ios.setf(ios_base:: skipws) ;</code> <code>ios.unsetf(ios_base:: skipws) ;</code>	I	Az elválasztó karakterek átlépése.
<i>unitbuf</i>	<code>ios.setf(ios_base:: unitbuf) ;</code> <code>ios.unsetf(ios_base:: unitbuf) ;</code>	O	Automatikus puffertöltés minden művelet után.
<i>uppercase</i>	<code>ios.setf(ios_base:: uppercase) ;</code> <code>ios.unsetf(ios_base:: uppercase) ;</code>	O	A számok megjelenítésben az (e és x) nagybetűk lesznek

Néhány adatfolyamot módosító manipulátor:

<i>Manipulátor</i>	<i>Megoldás tagfüggvényekkel</i>	<i>Írány</i>	<i>Leírás</i>
<i>endl</i>	<code>os.put(os, '\n') ; os.flush() ;</code>	O	Sorvége-karakter.
<i>ends</i>	<code>os.put('\0') ;</code>	O	Sztringvége-karakter.
<i>flush</i>	<code>os.flush() ;</code>	O	Puffertöltés
<i>ws</i>		I	Az elválasztó karakterek átlépése.

Az alábbi manipulátorok hatását az ugyanazon csoportban megadottal módosíthatjuk:

Manipulátor	Megoldás tagfüggvényekkel	Irány	Leírás
<i>A kiigazítás beállítása</i>			
left	os.setf(ios_base::left, ios_base::adjustfield);	O	Balra igazít.
right	os.setf(ios_base::right, ios_base::adjustfield);	O	Jobbra igazít – alapértelmezés
internal	os.setf(ios_base::internal, ios_base::adjustfield);	O	A mezőben a lehető legjobban széthúzva jeleknek meg az értékek.
<i>A számvrendszer beállítása</i>			
dec	os.setf(ios_base::dec, ios_base::basefield);	I/O	Tíz-es - alapértelmezés
hex	os.setf(ios_base::hex, ios_base::basefield);	I/O	Tizenhatos.
oct	os.setf(ios_base::oct, ios_base::basefield);	I/O	Nyolcas.
<i>A valós számok megjelenítése (hiányukban a fordító választ formátumot)</i>			
fixed	os.setf(ios_base::fixed, ios_base::floatfield);	O	Tizedes tört alakban.
scientific	os.setf(ios_base::scientific, ios_base::floatfield);	O	Hatványkitevős formában.

Paraméterekkel rendelkező manipulátorok

A paraméterrel rendelkező manipulátorok hatása általában csak a közvetlenül utánuk álló adatelemre terjed ki.

Manipulátor	Megoldás tagfüggvényekkel		Irány	Leírás
<code>setiosflags(lFlags)</code>	<code>long ios::flags(long lFlags);</code>	I/O	Jelzőbitek beállítása.	
<code>resetiosflags(lFlags)</code>	<code>long ios::unsetf(long lFlags);</code>	I/O	Jelzőbitek törlése.	
<code>setfill(nFill)</code>	<code>char ios::fill(char nFill);</code>	O	Kiülőtkarakter	
<code>setprecision(np)</code>	<code>int ios::precision(int np);</code>	O	Tizedesjegyek	
<code>setw(nw)</code>	<code>int ios::width(int nw);</code>	O	Mező-szélesség	
<code>setbase(nb)</code>	<code>...; os.setf(maszk, ios_base::basefield);</code>	O	Szám-rendszer.	

F2.2.3 I/O manipulátorok készítése

Saját manipulátorok készítését az alábbi példaprogrammal szemlélíthetjük:

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

// paraméter nélkül output manipulátor
ostream& tab(ostream& os) {
    os<<"\t";
    return os;
}

// paraméter nélkül input manipulátor
istream& urit(istream& is) {
    is.ignore(128, '\n');
    return is;
}

// tetszőleges saját manipulátor
class format {
private:
    int w, p;
friend ostream& operator<<(ostream& s, const format& m);
    format(int w=12, int p=7) : w(w), p(p) {}
};

ostream& operator<<(ostream& os, const format& m) {
    os.width(m.w);
    os.precision(m.p);
    os.setf(os::fixed, os::floatfield);
    return os;
}
```

```
void main() {
    cout<<setfill('*');
    cout<<setw(23);
    cout<<setprecision(4);
    cout<<"1.22379"<<endl;
    cout<<"1.22379"<<endl;
    cin.get();
}
```

```
*****1.224
1.224
```



```

int main() {
    int d;
    cout<<"Osszeg: ";
    cin>>d>>endl;
    cout<<"Az osszeg: "<<tab<<d<<endl;

    std::ios_fmtFlags sv=cout.flags(); // a beállítások mentése
    double pi=3.14159265;
    cout<<"pi="<<tab<<format("&>pi<<endl;
    cout<<"pi="<<tab<<format(7,5)<<pi<<endl;
    cout.flags(sv); // a beállítások visszatöltése
    cout<<"pi="<<pi<<endl;
    cin.get();
}

```

A program futásának eredménye:

```

Osszeg:1223
Az osszeg: 1223
pi= 3.1415927
pi= 3.14159
pi=3.1416

```

A példából jól látható, hogy a paraméter nélküli manipulator valójában egy speciális szerkezetű függvény, melynek címe adódik át a << és >> operátorok operandusaként. Ezzel szemben a paraméteres manipulator egy olyan osztály, melynek példánya jelenik meg a << és >> operátorok operandusában.

F2.2.4 Írás stringbe és olvasás stringből

Az *stringstream* fejeletben deklarált osztályok (*stringstream*, *ostringstream*, *stringstream*, *stringstream*, *stringbuf*) segítségével a karaktertípusokat adatfolyamként használhatjuk.

A konstans *string* objektumot a beviteli adatfolyamhoz az *istringstream* osztály konstruktorával rendeljük hozzá.

```

explicit istringstream (openmode mode = in);
explicit istringstream (const string& str, openmode mode = in);

```

Ha az adatkiviteli *ostringstream* adatfolyam esetében az alapértelmezett konstruktor használjuk, akkor egy belső stringpufferben dolgozhatunk, melynek tartalmát az *str()* függvény segítségével érhetjük el. Az opcionális paraméterrel eldönthetjük, hogy a string elejétől (ios::out) írjuk-e az adatokat, vagy pedig a tárolt karaktertípusokat végéhez (ios::ate) illesztjük az újabb szöveget.

```

explicit ostream (openmode mode = out);
explicit ostream (const string& str, openmode mode = out);

```

Az F.2.1.4. alfejezetben megoldott feladat objektumos változata:

```
#include <sstream>
#include <iomanip>
#include <iostream>
#include <string>
using namespace std;

int main()
{
    string adat[3] = {"22.7", "12.23", "1979.12"};
    double e;
    ostringstream sout;
    for (int i=0; i<3; i++) {
        istringstream sin(adat[i]);
        sin>>e; e += 0.5; // kerekítés
        sout.str("");
        sout<<setfill('0')<<setw(4)<<<long(e);
        adat[i] = sout.str();
    }
    for (int i=0; i<3; i++)
        cout<<i<<"\t "<<adat[i]<<endl;
}
```

F2.3 Állományok kezelése

Általános megfogalmazásban fájlok alatt a számítógép háttértaíraín (hájlékonylémez, merevélez, mágnesszalag, CD stb.) tárolt, névéel ellátott adatháalmazokat értünk. A fájlakat a bennük tárolt adatok elérése alapján csoportosíthatjuk. Azokat a fájlakat, amelyek adatelemeit csak sorban, egymás után érhetjük el, soros (szekvenciális) állományoknak nevezzük. A másik csoportot a közvetlen elérési fájlak alkotják. Ezekben a pozícionálás műveletét használva közvetlenül kiválaszthatjuk a feldolgozni kívánt adatelemeket.

F2.3.1 Állománytípusok

Az adatállományokat tartalmuk alapján szöveges és bináris fájlak csoportjára osztjuk. A szöveges állományokat általában soros fájlként használjuk, a bináris fájlak tartalmához pedig soros és közvetlen eléréssel egyaránt hozzáférhetünk.

Szöveges állományok

A szöveges állományok általában általunk is olvasható információit tartalmazznak. A szövegfájlok karaktereket tároló, különböző hosszúságú sorokból épülnek fel. Minden sor az *EOLN* (*End Of Line*) jel, az egész állományt pedig az *EOF* (*End Of File*) jel zárja (ami hiányozhat is a fájl végétől).

jel	vezérlőkarakterek	billentyűzettől
<i>EOLN</i> 13,10 (CRLF)	<i>EOF</i> 26	<i><Ctrl+Z></i>
		<i><Enter></i>

A szöveges állományok feldolgozása során az *EOF* karakter beolvasását a C++ alkalmazás fájlvége eseményként érzékeli. Mivel a *Unix* operációs rendszer alatt a szöveges állományokban a sorokat egyetlen újsor (*NL*, '\n') karakter zárja, a C++ alkalmazások a *Windows* rendszerben az állományok olvasása és írása során konverziót hajtanak végre. Amikor a C++ program szöveges állományba ír adatokat, akkor az *NL* karaktert két karakterre (*CR/LF*) alakítja át, illetve olvasáskor a fájlban található *CR/LF* karaktereket egyetlen *NL* karakterre cseréli.

Bináris állományok

A bináris fájlok bájtok sorozatát tárolják. Az állomány tartalmát tetszőleges sorszámból bájtól kezdve elérhetjük (a fájl első bájtjának sorszáma 0). Ellenkéntben a szöveges állományokkal, semelyik bájtérték sincs kitüntetve. (Természetesen a szövegfájlok mindig feldolgozhatók, mint bináris állományok.)

Alapvető különbség van a kettőle fájlípus között a különböző típusú számok tárolásában. Ha például a **short int** típusú 2005 számot szöveges állományba írjuk, akkor az olvasható formában, a '2', '0', '5' karaktertáblázatát tárolódik. Ugyanezt a számot bináris fájlba írva, a szám a memóriában tárolt formában (2 bájt) kerül az állományba.

F2.3.2 Szabványos C/C++ fájlkezelés

A számítógépeken az operációs rendszer feladatai közé tartozik a fájlrendszer támogatása és a fájlok elérésének biztosítása. Az állományok tartalmának eléréséhez a programozási nyelvtől és az operációs rendszer fájljától függetlenül mindig ugyanazokat a több lépéseket kell végrehajtani:

0. Előkészületek.
1. Az állomány megnyitása.
2. Az állomány tartalmának feldolgozása fájlműveletek (olvasás, írás, pozicionálás stb.) felhasználásával.
3. A szükséges műveletek elvégzése után az állomány lezárása.

A C nyelv állománykezelési lehetőségei közül csak a szabványos ún. adatfolyamkezelést tekintjük át, ahol a fájl egy **FILE** típusú struktúrára mutató pointer azonosítja:

```
#include <stdio>
using namespace std;
...
FILE *fp;
```

A fájl megnyitását a *fopen()* függvénnyel végezhetjük el, az állomány nevének és megnyitási módjának megadásával. A *modestring* (*mode*) betűivel kijelölhetjük a fájl típusát (*text*, *binary*) és elérési módját (*read*, *write*, *append*). Amennyiben írni és olvasni is szeretnénk az állományt a megnyitása után, az *rt+*, *w+* és *a+* jeleket kell használnunk.

```
FILE *fopen(const char *filename, const char * mode);
```

Ha a függvény által visszaadott érték nem *NULL*, használhatjuk a szokásos fájlműveleteket: írás, olvasás, pozicionálás. Minden műveletre függvények állnak rendelkezésünkre a szabványos függvénytárban:

<i>Fájlművelet</i>	<i>Függvény</i>
karakter írása	<code>int fputc(int c, FILE *stream);</code>
karakter olvasása	<code>int fgetc(FILE *stream);</code>
c-sztring írása	<code>int fputs(const char *s, FILE *stream);</code>
c-sztring olvasása	<code>char *fgets(char *s, int n, FILE *stream);</code>
formázott írás	<code>int fprintf(FILE *stream, const char *format, argument, ...);</code>
formázott olvasás	<code>int fscanf(FILE *stream, const char *format, address, ...);</code>
memóriaterület írása	<code>size_t fwrite(const void *ptr, size_t size, size_t n, FILE *stream)</code>
memóriaterület olvasása	<code>size_t fread(void *ptr, size_t size, size_t n, FILE *stream);</code>
pozicionálás	<code>int fseek(FILE *stream, long offset, int whence);</code>
pozíció lekérdezése	<code>long int ftell(FILE *stream);</code>
fájlvége vizsgálat	<code>int feof(FILE *stream);</code>
hibavizsgálat a műveletek után	<code>int ferror(FILE *stream);</code>

Munkánk végével le kell zárni az állományt, különben adatvesztés léphet fel:

```
int fclose(FILE *stream);
```

Az írási műveletek után, a lezárást megelőzően ajánlott az adatátviteli puffer tartalmát a fájlba üríteni:

```
int fflush(FILE *stream);
```

F2.3.3 A szabványos C++ állománykezelés

Az objektum-orientált fájlkézelés alapját a szabványos sablonkönyvtár általánosított adatfolyamosztályai (*stream*) képezik, melyek karakteralapú megvalósításait használjuk a fejezetben. Az osztályok deklarációinak eléréséhez az alábbi sorokat kell a programunkban elhelyezni:

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

A fájlkézelés során használható adatfolyamosztályokat táblázatban foglaltuk össze:

Általánosított osztály	Karakteralapú megvalósítás	Ismerető
<i>basic_istream</i>	<i>istream</i>	alaposztály
<i>basic_ostringstream</i>	<i>ostream</i>	input alaposztály
<i>basic_fstream</i>	<i>fstream</i>	output alaposztály
<i>basic_ifstream</i>	<i>ifstream</i>	input/output alaposztály
<i>basic_ofstream</i>	<i>ofstream</i>	input/output alaposztály
<i>basic_filebuf</i>	<i>filebuf</i>	output alaposztály
		alaposztály

Az adatfolyamosztályok alkalmazásának első lépése az objektumpéldány létrehozása és inicializálása. Már ebben a lépésben el kell döntenünk, hogy milyen irányú műveleteket kívánunk végezni: írni (*ofstream*), olvasni (*ifstream*) vagy mindkettőt (*fstream*). Az állomány megnyitása a konstruktorban is elvégezhető, de az *open()* tagfüggvény segítségével ezt később is megtehetjük. Alaphelyzetben adatfolyamok a szöveges fájlok kezelését támogatják. Amennyiben bináris állományt szeretnénk kezelni, meg kell adni az *ios::binary* konstanszt a fájlnyitás során. Az elmondottak alapján az adatfolyamobjektum létrehozása és az állomány nyitása:

```
ifstream ifstream::ifstream() ;
void ifstream::open(const char* szName, int nMode=ios::in,
                    int nProt = filebuf::openprot) ;
ifstream::ifstream(const char* szName, int nMode=ios::in,
                    int nProt = filebuf::openprot) ;
ofstream ofstream::ofstream() ;
void ofstream::open(const char* szName, int nMode=ios::out,
                    int nProt = filebuf::openprot) ;
ofstream::ofstream(const char* szName, int nMode=ios::out,
                    int nProt = filebuf::openprot) ;
fstream fstream::fstream() ;
void fstream::open(const char* szName, int nMode=ios::in,
                    int nProt = filebuf::openprot) ;
fstream::fstream(const char* szName, int nMode=ios::in,
                    int nProt = filebuf::openprot) ;
```

Munkánk végétével az adatfolyamot a *close()* tagfüggvény hívásával zárjuk.

```
fstream stream ("Proba.dat", ios::in | ios::out);
if (stream)
    cout<< "Sikeresi";
if (!stream)
    cout<< "Sikertelen";
if (stream.good())
    cout<< "Sikeresi";
if (stream.bad())
    cout<< "Hiba történt!";
if (stream.fail())
    cout<< "Súlyos hiba történt!";
if (stream.eof())
    cout<< "Fájlvége!";
stream.close();
```

A megnyitási (konstruálási) művelet, illetve a későbbiekben az adatfolyam-műveletek sikerességét (sikertelenségét) különböző módon ellenőrizhetjük:

<i>Konstans</i>	<i>Jelentés</i>
<i>fstream::openprot,</i> <i>fstream::sh_compat</i>	kompatibilis megosztási mód,
<i>fstream::sh_none</i>	kizárólagos mód – nincs megosztás,
<i>fstream::sh_read</i>	a megosztott olvasás engedélyezett,
<i>fstream::sh_write</i>	a megosztott írás engedélyezett.

Az *nProt* paraméter értéke, szintén előre definiált konstansokból, logikai vagy művelettel keletkezik (|).

<i>Konstans</i>	<i>fstream</i>	<i>ofstream</i>	<i>fstream</i>	<i>Jelentés</i>
<i>ios::app</i>		•	•	hozzáírás a fájl végétől,
<i>ios::ate</i>		•	•	pozicionálás a fájl végére,
<i>ios::in</i>	•	•	•	nyitás olvasásra,
<i>ios::out</i>		•	•	nyitás írásra,
<i>ios::trunc</i>			•	régi fájl tartalom eldobása,
<i>ios::nocreate</i>	•	•	•	csak létező fájl nyit meg,
<i>ios::noreplace</i>			•	csak új fájl nyit meg
<i>ios::binary</i>		•		bináris állomány.

A tagfüggvények argumentumaként az állomány nevét (*sName*), elérési módját (*nMode*) és védelmet (megosztását) (*nProt*) kell megadnunk. Az elérési módot az alábbi táblázatban szereplő konstansok felhasználásával, bitenkénti vagy (|) művelettel állíthatjuk elő:

Az adatfolyam sikeres megnyitása után különböző input-, illetve outputműveletek végzésére nyílik lehetőségünk. Az átdefiniált <> és << operátorok segítségével – alaptípusú adatokkal – csak szöveges adatfolyamot valósítható meg. A bináris és a szöveges fájlirtalomhoz tagfüggvények segítségével férhetünk hozzá. Az áttekintés során csak az alpműveletek bemutatására szorítkozunk.

Adatbeviteli műveletek végzése az istream tagfüggvényekkel

Az adatbeviteli műveletek többségének több tülterhelt változata is létezik a különböző karaktertípusokhoz, melyek közül csak a **char** típushoz készített változat prototípusát közöljük.

Tagfüggvény	
istream & operator >>alaptípusú_balérték;	szöveges adatbeolvasás,
int get();	karakter (bájt) olvasása,
istream& get(char& rch);	karakterisor (bájtsor) bevitele,
istream& get(char* pch, int ncount,	
char delim = '\n');	
istream& getline(char* pch, int ncount,	
char delim = '\n');	
istream& read(char* pch, int ncount);	bájtsor beolvasása.

Szükség esetén pozícionálhatunk is az állományban az abszolút, illetve a (fájl elejéhez – *ios::beg*, aktuális pozíciójához – *ios::cur* vagy végéhez viszonyított – *ios::end*) relatív pozíció megadásával:

```
istream& seekg( streampos pos );
istream& seekg( streamoff off, ios::seek_dir dir );
```

Az aktuális adatfolyam olvasási pozícióját kérdezhetjük le a *tellg()* tagfüggvénnyel:

```
streampos tellg();
```

Néhány további speciális művelet:

Tagfüggvény	
istream& ignore(int ncount = 1,	<i>ncount</i> darab bájt állépesse,
int delim = EOF);	a következő bájt tesztelése,
int peek();	az utolsó művelet során bevitt bájtok
int gcount() const;	száma,
istream& putback(char ch);	bájt visszairása az adatfolyamba.

Adatkiviteli műveletek végzése az ostream tagfüggvényekkel

Az adatkiviteli tagfüggvények csoportosítása az előző alfejezetben bemutatott műveletekhez hasonlóan végezhető el.

Tagfüggvény	
ostream& operator<<alaptípusú_jobbérték;	szöveges adatkivitel,
ostream& put(char ch);	karakter (bájt) kiírása,
ostream& write(const char* pch, int nCount);	bájt sor kivitele.

Az írási pozíció lekérdezéséhez, illetve beállításához külön tagfüggvényeket használhatunk:

```
streampos tellp();  
  
ostream& seekp( streampos pos );  
ostream& seekp( streamoff off, ios::seek_dir dir );  
  
Az adatkivitelre megnyitott adatfolyam lezárása előtt ajánlott üríteni az adatpuffereket:  
ostream& flush();
```


F3. A C-könyvtár legfontosabb elemei

Mielőtt rátérnénk a C++ nyelv sablonkonnyvtárának ismertetésére, röviden áttekinthetjük a szabványos ANSI C nyelvtől örökölt, leggyakrabban használt függvénycsoportokat. A fejezetben bemutatjuk a karakterek osztályozását és átalakítását, a számok és szövegek közötti konverziókat végző függvényeket, kitérünk a memória-, dátum- és időkezelő-kezelő megoldásokra valamint a matematikai rutinokra.

F3.1 Karakterek osztályozása és átalakítása

A C++ nyelv könyvtára gazdag készletet tartalmazza az egyetlen karaktert feldolgozó makróknak. A karakterkezelő makrókat és függvényeket a *cctype* deklarációs állománnyon keresztül érjük el. A C++ nyelv a karaktereket két csoportba osztja:

- A nyomtatható karaktereket
a számítógép meg tudja jeleníteni a képernyőn. A nyomtatható karakterek a szököz (0x20) és a hullámvonal (~, 0x7E) karakterek között helyezkednek el.
- Vezérlőkarakterek, melyek kódja a (0) és az (0x1F) intervallumba esik, de ide tartozik a Del (0x7F) karakter is.

A karaktertesztelő és karakter-átalakító függvények (makrók) **int** típusú paraméterrel rendelkeznek, melynek azonban csak az alsó bájtját használják. A széles-karakterrel (**wchar_t**) működő változatok nevében az *is/ito* után megjelenik a *w* betű, például: *iswalnum()*, *tolower()*.

int isalnum(int c) ;
Az *isalnum()* makró visszatérési értéke nem nulla, ha a *c* értéke betű ('A'-'Z', 'a'-'z') vagy számjegy ('0'-'9').

int isalpha(int c) ;
Az *isalpha()* makró visszatérési értéke nem nulla, ha a *c* értéke betű ('A'-'Z', 'a'-'z').

int iscntrl(int c) ;
Az *iscntrl()* makró visszatérési értéke nem nulla, ha a *c* értéke vezérlőkarakter.

int isdigit(int c) ;
Az *isdigit()* makró visszatérési értéke nem nulla, ha a *c* értéke számjegy ('0'-'9').

int isgraph(int c) ;
Az *isgraph()* makró visszatérési értéke nem nulla, ha a *c* nyomtatható karakter, de nem szököz.

int islower(int c) ;
Az *islower()* makró visszatérési értéke nem nulla, ha a *c* értéke kisbetű ('a'-'z').

int isprint(int c) ; Az *isprint()* makró visszatérési értéke nem nulla, ha a c nyomtatható karakter.

int ispunct(int c) ; Az *ispunct()* makró visszatérési értéke nem nulla, ha a c elválasztó karakter. Elválasztó karakternek számít az összes nyomtatható karakter a betűk, a számok és a szóköz nélkül.

int isspace(int c) ; Az *isspace()* makró visszatérési értéke nem nulla, ha a c szóköz, köcsivíssza, újsor, vízszintes és függőleges tabulátor vagy lapdobás karakter (0x09-0x0D, 0x20).

int isupper(int c) ; Az *isupper()* makró visszatérési értéke nem nulla, ha a c nagybetű ('A'-'Z').

int isxdigit(int c) ; Az *isxdigit()* makró visszatérési értéke nem nulla, ha a c hexadecimális számjegy ('0'-'9', 'A'-'F', 'a'-'f').

int tolower(int c) ; A *tolower()* függvény az angol nagybetűket kisbetűre alakítja. Visszatérési érték a c konvertált értéke.

int toupper(int c) ; A *toupper()* függvény az angol kisbetűket nagybetűre alakítja. Visszatérési érték a c konvertált értéke.

F3.2 Konverziós függvények

Az alábbi könyvtári függvények a különböző típusú numerikus értékek és karakteroszorozatok között végzik az átalakításokat. (Ere a célra természetesen a F2.1.4. fejezetben ismertett *sscanf()* és *sprintf()* függvényeket is használhatjuk.) A konverziós függvények prototípusát a *csdlib* fejlécfájl tartalmazza:

Karakteroszorozat átalakítása numerikus értéké

double atof(const char *s) ;

Az *atof()* függvény sztringet lebegőpontos (**double**) értéké konvertál. Ha az átalakítás sikertelen, a visszatérési érték 0.

int atoi(const char *s) ;

Az *atoi()* függvény karakteroszorozatot egész számmá alakít. Ha a konvertálás sikertelen, a visszatérési érték 0.

long atol(const char *s) ;

Az *atol()* függvény sztringet hosszú egész számmá konvertál. Ha a konvertálás sikertelen, a visszatérési érték 0L.

```
double strtod(const char *s, char **endptr);
double wcstod(const wchar_t *s, wchar_t **endptr);
```

A függvények az *s* sztringet duplapontosságú számmá alakítják. A függvények vissza-térési értéke a duplapontosságú szám. Ha az átalakítás hibás, akkor az **endptr* tartal-mazza a hibás karakterre mutató pointer. Tulcsordulás esetén a függvényérték a *HUGE_VAL*.

```
long strtol(const char *s, char **endptr, int radix);
long wcstol(const wchar_t *s, wchar_t **endptr, int radix);
```

Az *s* sztringet **long** értéké alakítják, és visszaadják függvényértékként. A *radix* a konvertálás alapszámát határozza meg (2-36). Ha *radix* 0, akkor az *s* sztring első néhány karaktere ('0', '0x') határozza meg a konvertálás alapját. Ha az átalakítás hibás, az **endptr* tartalmazza a hibás karakterre mutató pointer.

```
unsigned long strtoul(const char *s, char **endptr, int radix);
unsigned long wcstoul(const wchar_t *s, wchar_t **endptr, int radix);
```

A függvények az *s* sztring tartalmát **unsigned long** értéké alakítják. A visszatérési érték **unsigned long** típusú szám. Ha az átalakítás során hiba lép fel, a **endptr* paraméter a hibás karakterre mutató pointer tartalmazza.

Numerikus érték alakítása karakter sorozattá

```
char *ltoa(int value, char *string, int radix);
```

Az *ltoa()* függvény egész számot sztringgé alakít. A *value* értékét *EOS* karakterrel le-zárt sztringgé konvertálja a *string* mutatta tömbbe. A *radix* a konvertálás számszisz-temét határozza meg (2-36). Ha a *value* negatív, és a *radix* 10, akkor előjelesen konvertál, egyébként előjel nélkül. A visszatérési érték a sztringre mutató pointer.

```
char *ltoa(long value, char *string, int radix);
```

Az *ltoa()* függvény a *value* hosszú egész számot karakter sorozattá alakítja a *string* mu-tatta tömbbe. A *radix* a konvertálás számszisztemét határozza meg (2-36). Ha a *value* negatív, és a *radix* 10, akkor előjelesen konvertál, egyébként előjel nélkül. A visszaté-rési érték a sztringre mutató pointer.

```
char *ultoa(unsigned long value, char *string, int radix);
```

Az *ultoa()* függvény az **unsigned long** típusú *value* értéket sztringgé konvertálja. A *radix* a konvertálás számszisztemét határozza meg (2-36).

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Az *ecvt()* függvény a *value* lebegőpontos számot *ndig* számjegyet tartalmazó karakter-sorozatá konvertálja a tizedespon és az előjel elhelyezése nélkül. A **dec* változóba a tizedespon pozícióját teszi, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a statikus puffertben tárolt sztringre mutató pointer, amit az *fcvt()* és az *ecvt()* függvények következő hívása feleltér.

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Az *fcvt()* függvény a *value* lebegőpontos értéket *ndig* számjegyet tartalmazó sztringgé alakítja a tizedespon és az előjel elhelyezése nélkül. A **dec* változóba a tizedespon pozícióját teszi a rutin, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a statikus puffertben tárolt sztringre mutató pointer, amit az *fcvt()* és az *ecvt()* függvények következő hívása feleltér.

```
char *gcvt(double value, int ndec, char *buf);
```

A függvény a lebegőpontos *value* értéket karakter-sorozatá konvertálja tizedes tört alakban. A *value* számot, amennyiben lehetséges, *ndec* értékes számjegyet tartalmazó fixpontos (*FORTRAN F* formátumú) alakban kapjuk meg, egyébként lebegőpontos (*printf()* *E* formátumú) alakban. Az eredmény a *buf* mutatta puffertbe kerül, *EOS* karakterrel lezárva.

F3.3 Memóriaterületek (pufferek) kezelése

A puffer kifejezés alatt a memóriának adott címén kezdődő, adott hosszúságú területét értjük. A pufferek kezeléséhez szükséges függvények prototípusát a *cstring* állomány tartalmazza:

```
void *memchr(const void *s, int c, size_t n);
```

A *memchr()* függvény egy mutatót ad vissza, amely a *c*-ben adott karakter első előfordulásának helyét jelzi az *s* puffertben. (*NULL* jelzi, ha a karakter nincs benne.)

```
int memcmp(const void *s1, const void *s2, size_t n);
```

A *memcmp()* függvény összehasonlítja az *s1* és *s2* puffer első *n* bájtját. A visszatérési érték negatív, ha *s1* < *s2*, nulla, ha *s1* == *s2* és pozitív, ha *s1* > *s2*.

```
void *memcpy(void *dest, const void *src, size_t n);
```

A *memcpy()* függvény az *src* puffertől *n* bájtnyi blokkot átmásol a *dest* területre. Átfedésben levő területek esetén nem alkalmazható!

```
void *memmove(void *dest, const void *src, size_t n);
```

A *memmove()* függvény az *src* puffertől *n* bájtot átmozgat a *dest* területre. Egy mást átfedő pufferek esetén is jól működik.

```
void *memset(void *s, int c, size_t n);
```

A *memset()* függvény az *s* puffert első *n* bájtyát feltölti a *c* karakterrel, és *s* címével tér vissza.

F3.4 A dinamikus memóriakezelés függvényei

A dinamikus memóriakezelés elvégzéséhez szükséges függvények prototípusát a szabványos *csdlib* állomány tartalmazza.

```
void *malloc(size_t m);
```

A *malloc()* függvény *m* bájttal méretű memóriablokkot foglal le a halomterületen. A függvény visszatérési értéke az újonnan lefoglalt memóriablokkra mutató általános pointer.

```
void *calloc(size_t n, size_t m);
```

A *calloc()* függvény először lefoglal egy $n*m$ bájttal méretű memóriaterületet, majd pedig feltölti 0 értékű bájtokkal. A visszatérési értéke szintén a memóriablokkra mutató általános pointer.

```
void *realloc(void *ptr, size_t m);
```

A *realloc()* függvény újrafoglalással módosítja a már lefoglalt memóriablokk méretét. A *ptr* paraméter által kijelölt memóriablokk méretét *m* bájtra növeli vagy csökkenti. Ha a blokkot növelni kell, akkor a régi blokk tartalmát átmásolja az új helyre. A visszatérési érték az újonnan lefoglalt memóriablokkra mutató általános pointer. (Ha a *ptr* értéke *NULL*, akkor *malloc()* függvényként működik.)

Mindhárom függvény *NULL* (nulla) mutató visszatérési értékkel jelzi, ha nem sikerült elvégezni a kijelölt műveletet, ezért a programunkban mindig gondoskodnunk kell a függvényérték vizsgálatáról.

```
#include <csdlib>

void main()
{
    int *padat;
    padat = (int*) calloc (1223, sizeof(int));
    if (padat==NULL) exit (1);
    // ...
    free (padat);
}
```

Ha egy memóriablokkra nincs többé szükségünk, akkor azt fel kell szabadítanunk, átengedve annak területét a későbbi foglalási műveleteknek:

```
void free(void *ptr);
```

A *free()* függvény felszabadítja a *ptr* mutató által kijelölt memóriablokkot, amelyet előzőleg a *calloc()*, a *malloc()* vagy a *realloc()* hívások valamelyikével foglaltunk le.

Megjegyezzük, hogy a C++ alkalmazásokban általában a *new* és *delete* operátorokkal végezzük el a dinamikus memória foglalását és felszabadítását. Nem találunk azonban a *calloc()* és a *realloc()* függvények működésének megfelelő megoldást. Míg a dinamikus objektum példányok kezelésére csak az új műveleteket szabad használni, addig más esetekben (alaptípusú dinamikus tömbök stb.) a C-függvények is sikerrel alkalmazhatók.

F3.5 Matematikai függvények

A szabványos C-könyvtár különféle matematikai függvényeket tartalmaz, amelyek többsége **double** típusú paraméterrel és visszatérési értékkel rendelkezik. A függvények hibás működésüket a (*errno* állományban deklarált) *errno* változón keresztül jelzik:

EDOM	-	Domain error	-	Értelmezési tartomány hiba
ERANGE	-	Range error	-	Értékhatarát átlépés
HUGE_VAL	-	Overflow error	-	Túlcsordulás

A matematikai függvények használata esetén a *cmath* deklarációs állományt szükségsges a programunkba beépíteni:

Trigonometrikus függvények

```
double acos(double arg);
```

Az *acos()* függvény az *arg* árkusz koszinuszával tér vissza. A függvény argumentumak -1 és 1 közé kell esnie, máskülönben „domain error” hiba lép fel. A függvényértékek a 0 és π között helyezkednek el.

```
double asin(double arg);
```

Az *asin()* függvény az argumentum árkusz szinuszával tér vissza. Az argumentumnak -1 és 1 között kell lennie, máskülönben „domain error” hiba keletkezik. A függvényértékek $-\pi/2$ és $\pi/2$ között helyezkednek el. Az *acos()* és *asin()* függvények hibás argumentum-megadás esetén 0 értékkel térnek vissza, és az *errno* hibaváltozó értéke EDOM lesz.

double atan(double arg) ;

Az **atan()** függvény az *arg* arkusz tangensével tér vissza. A függvényértékek a $\pi/2$ és $\pi/2$ közé esnek.

double atan2(double y, double x) ;

Az **atan2()** függvény az y/x kifejezés arkusz tangensével tér vissza. A függvény felhasználja az argumentum előjelét a visszatérési érték tértünetének meghatározásához. A $-\pi$ és π között szolgáltatott értékek akkor is helyesek, amikor az eredmény szög $\pi/2$, illetve $-\pi/2$ közelében található (x 0-hoz közel van). Ha az x és az y értéke 0, akkor az **errno** változó értéke **EDOM** lesz.

double cos(double arg) ;

A **cos()** függvény az argumentum koszinuszával tér vissza. Az argumentum értékét radiánban kell megadni.

double sin(double arg) ;

A **sin()** függvény az *arg* szinuszával tér vissza. Az argumentum értékét radiánban kell megadni.

double tan(double arg) ;

A **tan()** függvény az *arg* tangensével tér vissza. Az argumentum értékét radiánban kell megadni. Ha az argumentum $\pi/2$ vagy $-\pi/2$ értékekhez közel van, a **tan** függvény 0 értékkel tér vissza, és az **errno** változó értéke **ERANGE** lesz.

Hiperbolikus függvények

double cosh(double arg) ;

A függvény az argumentum koszinusz hiperbolikus értékével tér vissza.

double sinh(double arg) ;

A függvény kiszámítja az argumentum szinusz hiperbolikus értékét.

double tanh(double arg) ;

A függvény megadja az argumentum tangens hiperbolikus értékét.

A **cosh()** és **sinh()** függvények túlcsoordulás esetén **HUGE_VAL** értékkel térnek vissza (előjel nélkül), és az **errno** felveszi az **ERANGE** értéket.

Hatvány- és logaritmusfüggvények

```
double exp(double arg) ;
```

Az *exp()* függvény az *e^{arg}* kifejezés értékével tér vissza. Tulcsordulás esetén a függvény értéke *HUGE_VAL*, míg alulcsordulás esetén *0.0* lesz. Mindkét esetet az *errno* változó *ERANGE* értéke jelzi.

```
double log(double num) ;
```

A *log()* függvény a *num* argumentum természetes alapú logaritmusával tér vissza. „Domain error” üzenetet kapunk, ha a *num* argumentum előjele negatív, és „Range error” hibáüzenetet kapunk, ha az argumentum zérus.

```
double log10(double num) ;
```

A *log10()* függvény kiszámolja az argumentum tízes alapú logaritmusát. Ha a *num* értéke *0* vagy negatív, akkor mindkét logaritmusfüggvény esetén az *errno* változó értéke *EDOM* lesz.

```
double pow(double x, double y) ;
```

A *pow()* függvény az *x* argumentum *y* hatványkifejezés emelt értékével (*x^y*) tér vissza. Tulcsordulás esetén a függvény értéke *HUGE_VAL*, és az *errno* változó értéke *ERANGE*. Ha az *x ≤ 0* és *y* nem egész, akkor *EDOM* hiba keletkezik, és a függvény értéke *HUGE_VAL* lesz. Ha *x* és *y* értéke *0 (0⁰)* a függvény értéke *1*.

```
double sqrt(double num) ;
```

Az *sqrt()* függvény a *num* négyzetgyökének értékével tér vissza. Ha az argumentum negatív előjele, akkor az *errno* változó értéke *EDOM* lesz, és *0.0* értékkel tér vissza a függvény.

```
double ldexp(double x, int exp) ;
```

Az *ldexp()* függvény kiszámítja az *x · e^{exp}* kifejezés értékét.

```
double frexp(double x, int *n) ;
```

A függvény meghatározza az *x* valós szám *m · eⁿ* alakú előállítását, ahol a mantissza (*m*) a függvény értéke (*0,5 ≤ m < 1*), *n* pedig a kitevő.

Egyéb cmath függvények

`double fabs(double x);`

Abszolút érték számítása.

`double ceil(double num);`

A függvény a legkisebb olyan egész számmal tér vissza, amelyik nem kisebb a *num* értéknél. Például $\text{ceil}(2.15) \rightarrow 3.0$, a $\text{ceil}(-2.15) \rightarrow -2.0$ értéket ad vissza.

`double floor(double num);`

A függvény a legnagyobb olyan egész számmal tér vissza, amely nem nagyobb *num* értéknél. Például $\text{floor}(2.15)$ esetén 2.0 és $\text{floor}(-2.15)$ esetén -3.0 lesz a visszatérési érték.

`double fmod(double x, double y);`

A függvény az *x modulo y*-t (az *x/y* osztás maradékát) adja vissza. A függvény az $x = iy + f$ kifejezést állítja elő, ahol az *i* egész szám, az $f(0 \leq f < y)$ pedig a függvény által visszaadott maradék.

`double modf(double x, double *ipart);`

A függvény két részre vágja az *x*-et: egy egésze, amelyet az *ipart*-ban tárol, és a törtészre, amit függvényértékként ad vissza.

A csdlib matematikai függvényei

`int abs(int num);`

A függvény az *int* típusú *num* abszolút értékével tér vissza.

`long labs(long num);`

A függvény a **long** típusú argumentum abszolút értékével tér vissza.

`div_t div(int x, int y);`

A függvény elosztja az argumentumként megadott két egész számot, és a *div_t* típusú struktúrában visszaadja a hányadost (*quot*) és a maradékot (*rem*).

`ldiv_t ldiv(long int x, long int y);`

A függvény elosztja az argumentumként megadott két hosszú egészet, és a *ldiv_t* típusú struktúrában visszaadja a hányadost (*quot*) és a maradékot (*rem*).

```
int rand(void);
```

A függvény alvételien számot ad vissza 0 és `RAND_MAX (0x7FFF)` között.

```
void srand(unsigned seed);
```

Az *srand()* függvény a pszeudovéletlen számok sorozatát inicializálja. Argumentum nélküli hívás esetén 1 lesz a kezdőérték, egyébként pedig a *seed* paraméter segítségével adhatjuk meg a kívánt értéket, mint az alábbi példában:

```
#include <cstdlib>
#include <ctime>
#include <iostream>

using namespace std;

void main(void) {
    srand(unsigned(0));
    // 1 és 90 közötti számok előállításá
    for(int i = 0; i < 12; i++)
        cout << rand() % 90 + 1 << "\t";
}
```

F3.6 Dátum- és időkezelő függvények

A dátum és az idő kezeléséhez a *ctime* deklarációs állományt kell beépíteni a programunkba. Az időkezelő függvények többsége a *tm* struktúrát használja:

```
struct tm {
    int tm_sec; // másodperc
    int tm_min; // perc
    int tm_hour; // óra (0+23)
    int tm_mday; // a hónap napja (1+31)
    int tm_mon; // hó (0+11)
    int tm_year; // év (a nap társított év - 1900)
    int tm_wday; // a hét napja (0+6; vasárnap = 0)
    int tm_yday; // az év napja (0+365)
    int tm_isdst; // 0 ha nincs nyári időszámítás
};
```

```
char *asctime(const struct tm *tblock);
```

A *asctime()* függvény 26-karakteres karaktertárolóban adja vissza a **tblock* struktúrában tárolt dátumot és időt (a 26. karakter az *EOS*). A sztringet a következő *asctime* vagy *ctime* hívás felülírja. A sztring felépítése: `Sun Dec 23 07:12:23 1979\n\0`.

```
clock_t clock(void);
```

A *clock()* függvény segítségével két esemény közötti időintervallum határozható meg. Ha az időértékeket másodpercenként kívánjuk megkapni, a visszaadott értéket el kell osztani a *CLK_TCK* konstanssal.

```
char *ctime(const time_t *time) ;
```

A *ctime()* függvény a dátumot és az időt karaktertorsorozattá alakítja, majd visszatér a 26-karakteres sztringre mutató pointerrel. A **time* paraméter értéke a *time()* függvény hívásával állítható be.

```
double difftime(time_t time2, time_t time1) ;
```

A *difftime()* függvény a megadott két *time_t* típusú időpont különbségét határozza meg másodpercekben.

```
struct tm *gmtime(const time_t *timer) ;
```

A *gmtime()* függvény az aktuális dátumot és az időpontot Greenwich-i idővé (GMT) alakítja, és egy *struct tm* típusú struktúrába tölti. (Ez a statikus struktúra minden híváskor felülíródik.)

```
struct tm *localtime(const time_t *timer) ;
```

A *localtime()* függvény a (helyi) *time_t* típusú dátumot és az időpontot egy *struct tm* típusú struktúrába tölti. (Ez a statikus struktúra minden híváskor felülíródik.)

```
time_t mktime(struct tm *t) ;
```

Az *mktime()* függvény kitölti a *tm* struktúra hiányzó adattagjait, majd *time_t* formátúra alakítja a struktúrában tárolt időt és dátumot.

```
time_t time(time_t *timer) ;
```

A *time()* függvény visszatérési értéként a nem *NULL timer* paraméterrel kijelölt területen megadja az aktuális, az 1970. január 1. 00:00:00 óra GMT óta eltelt időt másodpercekben.

```
size_t strftime(char *str, size_t maxsize,
const char *format, const struct tm *t) ;
```

A függvény a *tm* típusú struktúrában tárolt (*t) időpontot legfeljebb *maxsize* hosszú sztringgé alakítja a *format* formátum előírásai alapján. A formátumelemeket az alábbi táblázatban foglaltuk össze, magyar helyi beállítások – *setlocale(LC_ALL, "Hun")*; – használatára esetén

<i>Elem</i>	<i>Magyarázat</i>	<i>Példa</i>
%a, %A	rövid és hosszú napnév	cs, csütörtök
%b, %B	rövid és hosszú hónapnév	dec. december
%c	dátum és idő	2004.12.23. 12:23:07
%x	dátum	2004.12.23.
%X	idő	12:23:07
%#c	dátum és idő	2004. december 23. 12:23:07

Elem	Magyarázat	Pelida	(folytatás)
%%#x	dátum	2004. december 23.	
%d	a hónap hányadik napja	01 - 31	
%H	24-órás idő órája	00 - 23	
%I	12-órás idő órája	01 - 12	
%j	az év hányadik napja	001 - 366	
%m	a hónap sorszáma	01 - 12	
%M	a percek száma	00 - 59	
%p	a 12-órás idő de/du jelzője	de., du.	
%S	a másodpercek száma	00 - 59	
%u	hányadik hete az évnek (vasárnap a hét első napja)	00 - 53	
%w	hányadik napja a hétnek	0 - 6; vasárnap 0	
%W	hányadik hete az évnek (hétő a hét első napja)	00 - 53	
%Y	év az évszázad nélkül	00 - 99	
%y	év az évszázaddal	2004	
%Z, %z	az időzóna teljes neve	Közép-Európa - téli idő	
%%	százalékjel	%	

Az alábbi elemek használatakor a kezdő nullák nem jelennek meg: %d, %H, %I, %#, %m, %M, %S, %U, %W, %Y. Az alábbi kis program jól szemlélteti a fenti függvények használatát:

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

void main()
{
    char puffer[128];
    time_t ltime;
    struct tm ma,
    *gmt, husvet = { 07, 23, 11, 27, 2, 105 };
```

A Unix ido:	Sat Dec 25 23:12:09 2004
A Greenwich-i ido:	Sat Dec 25 22:12:09 2004
Helyi ido:	Sat Dec 25 23:12:09 2004
2005 Husvet:	Sun Mar 27 12:23:07 2005
2004. december 25. 23:12:09	2004.12.25. 23:12:09

```
time(&ltime);
cout<<"A Unix ido:\t\t" << ctime(&ltime) <<endl;
gmt = gmtime(&ltime);
cout<<"A Greenwich-i ido:\t" << asctime(gmt) <<endl;
ma = *localtime(&ltime);
cout<<"Helyi ido:\t\t" << asctime(&ma) <<endl;
ltime=mktime(&husvet);
cout<<"2005 Husvet:\t\t" << ctime(&ltime) <<endl;
```

```
// Az idősztring magyar helyi beállításokat használva
setlocale(LC_ALL, "Hun");
strftime(buf, 128, "%c %nc", &ma);
cout<< buf<<endl;
cin.get();
}
```

F3.7 Rendezés és keresés

A C-könyvtár néhány hasznos algoritmus megvalósítását is tartalmazza. Az alábbi rendező és kereső függvények prototípusa a *csdlib* fejállományban található.

A quicksort algoritmust megvalósító sorda rendező függvény

```
void qsort(void *base, size_t nemlem, size_t width,
           int(*fcmp)(const void *, const void *));
```

A *qsort()* függvény gyorsrendezést hajl végre a *base* mutató által kijelölt tömbben. A függvény további paramétereinek jelentése:

nemlem az elemek száma a tömbben,
width egy elem mérete byte-okban kifejezve,
fcmp az összehasonlító függvényre mutató pointer.

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

```
int fcmp(const void * p, const void * q)
{
    return *(const int *)p - *(const int *)q;
}
```

```
void main()
{
    int a[7] = {07, 23, 12, 79, 25, 10, 02};
    qsort(a, 7, sizeof(int), fcmp);
    for (int i=0; i<7; i++)
        cout<<a[i]<<"\t";
    cin.get();
}
```

Bináris keresés elemek rendezett tömbjében

```
void * bsearch(const void *key, const void *base,
               size_t nemlem, size_t width,
               int(*fcmp)(const void *, const void *));
```

A *bsearch()* függvény a **key* elemet keresi a rendezett *base* tömbben. A függvény visszatérési értéke az elsőnek megtalált elemre mutató pointer, vagy *NULL*, ha nem

találta meg. A paraméterek jelentése megegyezik a *qsort()* függvény azonos nevű paramétereinek értelmezésével. Az összehasonlító függvény visszatérési értéke:

<0,	ha *elem1	<	*elem2,
0,	ha *elem1	==	*elem2,
>0,	ha *elem1	>	*elem2.

F3.8 Egyéb hasznos függvények

```
#include <stdlib.h>
void exit (int status);

A függvényhívás megszakítja a C++ program futását. Az argumentumban a kilépési
kódot adhatjuk meg: EXIT_SUCCESS (0) hibátlan programbefejezés, EXIT_FAILURE
(1) hiba miatti kilépés.
```

```
#include <cassert>
void assert(int expression);
```

A *assert()* makró kiértékeli az argumentumaként megadott kifejezést, és amennyiben az 0 értékű, hibaüzenettel megszakítja a program futását, például:

```
Assertion failed: 0, file c:\progcpp\fs\nata.cpp, line 23
```

A makró nem működik, ha a *cassert* deklarációs állomány beépítése előtt definiáljuk az *NDEBUG* szimbólumot.

```
#include <stdlib.h>
void abort(void);
```

Az *abort()* függvény megszakítja az éppen futó folyamatot, és megjelenti az „*abnormal program termination*” üzenetet.

```
#include <stdlib.h>
int system(const char * command);
```

A futó program aktivizálja az operációs rendszer parancsértelmezőjét, és átadja neki a *command* paramcsot. A függvény 0 visszatérési értéke a sikeres parancsvégrehajtást jelzi, míg a -1 hibára utal, melynek kódját az *errno* változóban találjuk.

```
#include <locale>
char *setlocale (int category, const char *locale);
```

A helyi (országfüggő) beállítások kategóriáit (*LC_COLLATE*, *LC_CTYPE*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME* vagy mindet egyszerre *LC_ALL*) állíthatjuk be, illetve kérdezhetjük le. Beállításnál a *locale* karaktertensorozat határozza meg az országot ("C", "Hung", "English", "German", "Rus" stb.), míg a kategória leképezésénél értéke *NULL*.

F4. A Szabványos sablonkönyvtár (STL) elemei

A Hewlett-Packard Company által fejlesztett Szabványos sablonkönyvtár (STL, *Standard Template Library*) a C++ szabványos könyvtára, amely általánosított (sablon-) osztályokat és függvényeket tartalmaz a leggyakrabban használt adatstruktúrák és algoritmusok megvalósításaként. A könyvtár az általánosított (*generic*) programozást konténerekkel (tárolókkal), iterátorokkal (általánosított mutatókkal) és algoritmusokkal támogatja. Az alábbiakban rövid áttekintést adunk arról a három összetevőről. Az STL elemeit és az eléréshez szükséges deklarációs állományokat az alábbi táblázat tartalmazza:

Rövid leírás	Fejállomány
Adatsorkezelés, rendezés, keresés stb. (STL)	<algorithm>
Általánosított numerikus műveletek	<numeric>
Asszociatív tároló: bithalmaz (<i>bitset</i>)	<bitset>
Asszociatív tároló: halmazok (elemismétlődéssel – <i>multiset</i> , illetve ismétlődés nélkül – <i>set</i>)	<set>
Asszociatív tároló: kulcs/érték adatpárok tárolása 1:1 (<i>map</i>), illetve 1:n (<i>multiset</i>) kapcsolatban (leképzések)	<map>
Műveletobjektumok	<functional>
Iterátorelemek, előre definiált iterátorok, adattípus-iterátorok	<iterator>
Memóriakezelés	<memory>
Műveleti elemek	<utility>
Tároló: dinamikus tömb (<i>vector</i>)	<vector>
Tároló: kettős végű sor (<i>deque</i>)	<deque>
Tároló: lineáris lista (<i>list</i>)	<list>
Tároló: sor (<i>queue</i>), prioritásos sor (<i>priority_queue</i>)	<queue>
Tároló: verem (<i>stack</i>)	<stack>

F4.1 Tárolók (konténerek)

A tárolókat két fő csoportba sorolhatjuk: adatsorok (soros) és asszociatív tárolók. A soros tárolókra (vektorok, listák, sorok stb.) jellemző, hogy rendezettek, az elemek sorrendje által. Az asszociatív tárolók (leképzés, halmaz stb.) közös tulajdonsága, hogy az elemeket egy kulcs alapján lehet elérni.

A tárolóinterfészek általában az alábbi összetevőkből épülnek fel: konstruktorok (az alapértelmezett és a másoló konstruktor is), elem elérése, elem beszűrése, elem törlése, destruktör és iterátorok. A tárolókat iterátorok segítségével kezeljük. Az iterátor olyan „mutatószerű” objektum, amelyet az STL tárolókhoz hoztak létre. Példaként tekintsük az alábbi programot, ahol egy vektor elemeit függvény segítségével összegezzük!

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

double osszeg(const vector<double> & dv)
{
    vector<double>::const_iterator p; // konstans iterátor
    for (p=dv.begin(); p!=dv.end(); p++)
        s += *p;
    return s;
}

int main()
{
    ostream_iterator<double> out(cout, " "); // kimeneti iterátor
    vector<double> v(4,0);
    copy(v.begin(), v.end(), out); // a tömb kilírása
    cout<<"Elemösszeg: "<<osszeg(v)<<endl;
    // A program futásának eredménye: 7 12 23 79 Elemösszeg: 121
    cin.get();
}
```

Az alábbi táblázatban összefoglaltuk a CC-vel jelölt konténerosztályok interfészeit:

<i>Típus</i>	<i>Leírás</i>
CC::container_type	a konténer típusa,
CC::value_type	a konténerben tárolt adatok típusa,
CC::size_type	a konténer méretének típusa,
CC::reference	a tárolt értékek referenciájának típusa,
CC::const_reference	konstans referencia
CC::pointer	mutató a referenciátípushoz,
CC::iterator	iterátor típus,
CC::const_iterator	a konstans iterátor típusa,
CC::reverse_iterator	a fordított iterátor típusa,
CC::const_reverse_iterator	a konstans fordított iterátor típusa,
CC::difference_type	két CC::iterator típusú érték különbségének típusa.

Minden egyes tároló rendelkezik a fenti típusokkal. Például, egész elemű vektor `vector<int>` esetén az elemek típusa `vector<int>::value_type`, a bejáráshoz pedig a `vector<int>::iterator` típussal készíthetünk iterátort.

Az alábbi táblázat a leggyakrabban használt konténeretagekokat tartalmazza. (Míg `CC` a konténer osztályát jelölte, addig a `c` a konténerobjektum.

Tagfüggvény	Leírás
<code>CC::CC()</code>	alapértelmezett konstruktor,
<code>CC::CC(c)</code>	másoló konstruktor,
<code>c.begin()</code>	a <code>c</code> konténer első eleme (kezdet),
<code>c.end()</code>	a <code>c</code> tároló utolsó eleme utáni pozíció (vége),
<code>c.rbegin()</code>	kezdet a fordított iterátor számaára,
<code>c.rend()</code>	vég a fordított iterátor számaára,
<code>c.size()</code>	a konténerben tárolt elemek száma,
<code>c.max_size()</code>	a legnagyobb elemszám,
<code>c.empty()</code>	igaz, ha a tároló üres,
<code>c.swap(d)</code>	két tároló elemének felcserélése.

A konténerekhez műveleti jelek is tartoznak, amelyek `CC::value_type` típust használják: `==`, `!=`, `<`, `>`, `<=`, `>=`.

Soros konténerek

Soros tárolók (*vector*, *list* és *deque*) esetén az elemeket sorban, egymás után is elérhetjük. Az esetek többségében ide sorolható a `C++` nyelv tömbtípusa is. Az alábbi táblázatban összefoglaltuk a soros (*sequence*) konténerek (*SEQ*) specifikus tagfüggvényeit:

Tagfüggvény	Leírás
<code>SEQ::SEQ(n, v)</code>	konstruálás <code>n</code> darab <code>v</code> értékű elemmel,
<code>SEQ::SEQ(b_it, e_it)</code>	konstruálás a megadott iterátorok közötti $(b_it, e_it - 1)$ elemek átmásolásával,
<code>c.insert(w_it, v)</code>	<code>v</code> beszúrása a <code>w_it</code> elé,
<code>c.insert(w_it, v, n)</code>	<code>n</code> darab <code>v</code> érték beszúrása a <code>w_it</code> elé,
<code>c.insert(w_it, b_it, e_it)</code>	b_it és e_it közötti elemek beszúrása <code>w_it</code> elé,
<code>c.erase(w_it)</code>	a <code>w_it</code> által kijelölt elem törlése,
<code>c.erase(b_it, e_it)</code>	a b_it és e_it közötti elemek törlése.

Asszociatív tárolók

Asszociatív tárolók (*set*, *map*, *multiset* és *multimap*) esetén az elemeket kulcs (*key*) segítségével érjük el. Ezek a konténerek rendelkeznek a *Compare* rendezési relációval, amely valójában egy összehasonlító objektum. Az asszociatív konténerek használata az alábbi példa szemlélteti:

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

using namespace std;

void main()
{
    map<string, int, less<string> > nev_kor;

    nev_kor["Natalia"] = 12;
    nev_kor["Lafénitla"] = 25;
    nev_kor["Nata"] = 23;
    cout<<"Lafénitla "<<nev_kor["Lafénitla"]<<" éves."<<endl;
    // A program kimenete: Lafénitla 25 éves.
    cin.get();
}
```

Az asszociatív tárolók osztályainak (ASSOC) interfészei kibővítik a CC-vel jelölt konténerosztályok interfészét:

Leírás	Típus
ASSOC::key_type a kereső kulcs típusa,	ASSOC::key_compare az összehasonlító objektum típusa,
az ASSOC::value_type típusú értékek összehasonlításához használt típus.	ASSOC::value_compare az összehasonlító objektum típusa,

Az asszociatív tárolók több szabványos konstruktorral is rendelkeznek. Az asszociatív konténerek konstruktora elsősorban az összehasonlító objektum használataiban különbözik a soros tárolók konstruktorától.

Leírás	ASSOC()	ASSOC(cmp)	ASSOC(b_it, e_it, e_it)	ASSOC(b_it, e_it, cmp)
a <i>Compare</i> összehasonlító objektumot használó alapértelmezett konstruktor,				
a <i>cmp</i> összehasonlító objektumot használó konstruktor,				
konstruálás a <i>b_it</i> és <i>e_it</i> közötti elemekből a <i>Compare</i> felhasználásával,				
konstruálás a <i>b_it</i> és <i>e_it</i> közötti elemekből a <i>cmp</i> összehasonlító objektum felhasználásával,				

A törlés és a beszúrás műveletek is kibővíthetők a kulcs alkalmazásának következtében:

Tagfüggvény	Leírás
<code>c.insert(c)</code>	<code>t</code> beszúrása, ha egyetlen elem sem rendelkezik <code>t</code> kulcsával; a visszaadott <code>pair<iterator, bool></code> értékben a <code>bool</code> értéki, ha a <code>t</code> nem volt benne,
<code>c.insert(w_it, c)</code>	<code>t</code> beszúrása a <code>w_it</code> kezdőpozíciótól; <code>set</code> és <code>map</code> esetén nem végzi el a műveletet, ha a kulcs már használt; a beszúrás pozíciójával tér vissza,
<code>c.insert(b_it, e_it)</code>	<code>b_it</code> és <code>e_it</code> közötti elemek beszúrása, törli a <code>k</code> kulcsú elemeket, és visszaadja a törlött elemek számát,
<code>c.erase(k)</code>	<code>c.erase(w_it)</code>
<code>c.erase(b_it, e_it)</code>	a <code>w_it</code> által kijelölt elem törlése, a <code>b_it</code> és <code>e_it</code> közötti elemek törlése.

További hasznos tagfüggvények:

Tagfüggvény	Leírás
<code>c.find(k)</code>	a <code>k</code> kulcsú elemre hivatkozó iterátorral tér vissza,
<code>c.count(k)</code>	megadja a <code>k</code> kulcsú elemek számát,
<code>c.lower_bound(k)</code>	megadja az első olyan elem iterátorát, amelyik kulcsa nagyobb vagy egyenlő, mint <code>k</code> ,
<code>c.upper_bound(k)</code>	megadja az első olyan elem iterátorát, amelyik kulcsa kisebb vagy egyenlő, mint <code>k</code> ,
<code>c.equal_range(k)</code>	visszaadja a <code>(lower_bound(k), upper_bound(k))</code> iterátort.

Konténeradapterek

A konténeradapter-osztályok, olyan konténerosztályok, amelyek módosítják az alapvető tároló osztályokat, a létező megvalósításától eltérő viselkedés biztosítása érdekében. A támogatott konténeradapterek: *stack*, *queue*, *priority_queue*.

A „*last-in, first-out*” működésű *stack* egyaránt adaptálható a *vector*, a *list* és a *deque* tárolókból. Az adaptált *stack*-függvényeket táblázatban foglaltuk össze:

Tagfüggvény	Leírás
<code>void push (const value_type& v)</code>	v bevitele a verembe,
<code>void pop ()</code>	a verem felső elemének levétele,
<code>value_type& top ()</code>	a verem felső elemének elérése,
<code>const value_type& top () const</code>	a verem felső elemének lekérdezése,
<code>bool empty () const</code>	true érték jelzi, ha a verem üres,
<code>size_type size () const</code>	a veremben lévő elemek száma,
<code>operator== és operator<</code>	egyenlőség és lexikografikus ki-
	sebb művelet.

Az alábbi példaprogramban adott számszámrendszerbe való átváltásra használjuk a vermet:

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

void main() {
    int szam=2005, alap=16;
    stack<int> v;
    while (szam>0) {
        v.push(szam % alap);
        szam /= alap;
    }
    while (!v.empty()) {
        szam = v.top();
        v.pop();
        cout<<(szam<10 ? char(szam+'0') : char(szam+'A'-10)) << " ";
        cin.get();
    }
}
```

A „first-in, first-out” működésű sort (*queue*) *list* vagy *deque* tárolókból adaptálhatjuk. Az adaptált *queue*-függvényeket az alábbi táblázat tartalmazza:

Tagfüggvény	Leírás
<code>void push (const value_type& v)</code>	v hozzákapcsolása a sor végéhez,
<code>void pop ()</code>	a sor első elemének levétele,
<code>value_type& front ()</code>	a sor első elemének elérése,
<code>const value_type& front () const</code>	a sor első elemének lekérdezése,
<code>value_type& back ()</code>	a sor utolsó elemének elérése,
<code>const value_type& back () const</code>	a sor utolsó elemének lekérdezése,
<code>bool empty () const</code>	true értékkel jelzi, ha a sor üres,
<code>size_type size () const</code>	a sorban található elemek száma,
<code>operator== és operator<</code>	egyenlőség és lexikografikus ki-
	sebb művelet.

A prioritásos sor (*priority_queue*) a *vector* vagy a *deque* tárolóból adaptálható. Szükségünk van egy összehasonlító objektumra is, hisz az elemek rendezetten helyezkednek el – a legfelső elem a legnagyobb. Az adaptált *priority_queue*-függvényeket az alábbi táblázatban foglaltuk össze:

<i>Tagfüggvény</i>		<i>Leírás</i>
<code>void push (const value_type& v)</code>	a bevitele a prioritásos sorba,	
<code>void pop ()</code>	a prioritásos sor felső elemének le-	
<code>const value_type& top () const</code>	a prioritásos sor felső elemének vétele,	
<code>bool empty () const</code>	lekérdezése,	
<code>true</code> értékkel jelzi, ha a prioritásos sor üres,	megadja a prioritásos sorban talál-	ható elemek számát,
<code>size_type size () const</code>		

F4.2 Iterátorok

A tárolókban iterátorok segítségével „közlekedhetünk”. Az iterátorokat öt csoportba sorolhatjuk:

<code>istream_iterator</code>	<code>input</code>	bemeneti,
<code>ostream_iterator</code>	<code>output</code>	kimeneti,
<code>iterator</code>	<code>forward</code>	előre haladó (az előző kettő is ilyen),
<code>bidirectional_iterator</code>	<code>bidirectional</code>	kétirányú,
<code>random_access_iterator</code>	<code>random access</code>	közvetlen hozzáféréssel.

Az *istream_iterator* és az *ostream_iterator*

A kimeneti és a bemeneti iterátorok segítségével a tároló elemei és az I/O adatfolyamok közötti létesítünk kapcsolatot.

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

void main()
{
    double x[5] = {7, 12, 23, 79, 157};
    ostream_iterator<double> out(cout, "\\t");
    for (int i=0; i<5; i++)
        *out=x[i];
    cout<<endl;
    copy(x, x+5, out);
    cout<<endl;
```



```

void main() {
    int x[5]={7, 79, 12, 23, 123};
    ostream_iterator<int> out(cout, "t");
    vector<int> v(x, x+5);
    copy(v.begin(), v.end(), out);
    cout<<endl;
    reverse_iterator p=v.rbegin();
    copy(p, v.end(), out);
    cout<<endl;
}

```

7	79	12	23	123
19	79	12	23	123
123	23	12	79	19

F4.3 Algoritmusok

A különböző algoritmusokat függvénysablonok formájában tartalmazza az STL. Az algoritmusokat négy csoportba sorolhatjuk:

- elemssorrendet nem változtató algoritmusok,
- elemssorrendet megváltoztató algoritmusok,
- numerikus algoritmusok.

Az alábbiakban az egyes algoritmusoknak csak a nevét adjuk meg, hisz teljes ismeret-
tőlük külön igényelne.

Az elemssorrendet nem változtató algoritmusok:

accumulate	adjacent_find	includes
binary_search	count_min	lower_bound
count_if	max	max_element
equal	min	min_element
equal_range	mismatch	nth_element
find	search	search_n
find_end	find_first_of	
find_if		
for_each		

Az elemsorrendet megváltoztató algoritmusok:

copy	remove_if
copy_backward	fill
fill	fill_n
generate	generate_n
generate_n	inplace_merge
iter_swap	rotate
make_heap	rotate_copy
merge	set_difference
nth_element	set_symmetric_difference
next_permutation	set_intersection
partial_sort	set_union
partial_sort_copy	sort
partition	sort_heap
prev_permutation	stable_partition
push_heap	stable_sort
pop_heap	swap
random_shuffle	swap_ranges
remove	transform
remove_copy	unique
remove_copy_if	unique_copy

Numerikus algoritmusok

inner_product
partial_sum
adjacent_difference

Az alábbi példaprogramban a különböző algoritmusok tömbre való alkalmazását követhetjük nyomon:

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

void main()
{
    int adat[5]={5, 7, 12, 23, 46};
    ostream_iterator<int> out(cout, "\t");
    // az adatok véletlenszerű keverése:
    for (int i=0; i<7; i++)
        random_shuffle(adat,adat+5);
    copy(adat,adat+5, out);
    cout<<endl;
}
```



```
// az adatok rendezése:
sort(adat,adat+5);
copy(adat,adat+5, out);
cout<<endl;

// adatok felcserélése:
swap(adat[2],adat[4]);
copy(adat,adat+5, out);
cout<<endl;

// tömb felöltése 23-mal
fill(adat,adat+5, 23);
copy(adat,adat+5, out);
cout<<endl;
cin.get();
```

46	7	23	23	12	5
46	23	12	7	5	7
7	5	23	7	5	46
12	23	7	5	23	12
12	46	7	5	5	46
12	46	12	23	5	23
46	7	7	23	5	7
5	5	46	12	46	12
23	23	23	23	23	23

F4.4 Műveltetőjekümmök (functions)

A különböző műveleteket lefedő műveletobjektumok definíciójak az *operator* (függvény). A műveletobjektumoknak fontos szerepük van az általánosított algoritmusok használatában, hiszen ezen algoritmusok interfésze vagy egy függvénymutató, vagy pedig egy olyan objektumot fogad, amely definíciójá az *operator* (függvény). A *(functional)* könyvtárban megtalálható műveletobjektumokat három csoportba oszt-hatjuk:

Aritmetikai műveletek

$x + x$	$0sszeadas$
$x - x$	$kivoias x - y$
$x * x$	$szozas$
x / x	szo
$x \% x$	$maradek$
$x -$	sza

Relaci3n m3vlelek

equal	$x == y$
not-equal	$x \neq y$
greater	$x > y$
less	$x < y$
greater-equal	$x \geq y$
less-equal	$x \leq y$
negate	$\neg x$

Logikai műveltség

```
logical_and  
logical_or  
logical_not  
logikai és x & y  
logikai vagy x || y  
logikai tagadás / x
```

Az alábbi példában saját műveleti objektumot definiálunk a faktoriális-számítás elvégzésére:

```
#include <functional>
#include <deque>
#include <vector>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

template<class Arg>
class faktoriális : public unary_function<Arg, Arg> {
public:
    Arg operator()(const Arg& arg) {
        Arg a = 1;
        for (Arg i = 2; i <= arg; a*=i++);
        return a;
    }
};

int main() {
    int init[5] = {3,5,7,11,12};
    deque<int> d(init, init+5);
    // üres vektor létrehozása
    vector<int> v(size_t{5});
    // a kétvégű sorban tárolt elemek átalakítása a faktoriális
    // felhasználásával, és tárolása a vektorban
    transform(d.begin(), d.end(), v.begin(), v.begin(), faktoriális{});
    // megjelentés
    ostream_iterator<int> out(cout, "\t");
    copy(d.begin(), d.end(), out);
    cout << endl;
    copy(v.begin(), v.end(), out);
    cin.get();
}
```

3	5	7	11	12
6	120	5040	39916800	479001600

Műveletadapterek (function adaptors)

A műveletadapterek segítségével műveletobjektumokat hozhatunk létre adaptáció révén.

not1	az átadott egyoperandusú predikátum tagadásával (!) tér vissza
not2	az átadott kétoperandusú predikátum tagadásával (!) tér vissza
binder1st	a műveletobjektum 1. operandusának kijelölése,
binder2nd	a műveletobjektum 2. operandusának megadása,
ptr_fun	mutató létrehozása egy-, illetve kétoperandusú műveletobjektumhoz.

Az alábbi példában a *bind2nd* adapter segítségével olyan összeadó műveletobjektumot készítettünk, amely 23-at ad az első operandusához:

```
#include <functional>
#include <algorithm>
#include <iterator>
#include <iostream>
using namespace std;

int main()
{
    double adat[5] = { 3, 5, 7, 11, 12 };
    transform(adat, adat+5, adat, bind2nd(plus<double>(), 23));
    ostream_iterator<double> out(cout, "\\t");
    copy(adat, adat+5, out);
    cout << endl;
    cin.get();
}
```

26	28	30	34	35
----	----	----	----	----

F4.5 Allokátorok (allocators)

Az allokátorobjektumok (*memory*) a tárolók számára lefoglalt memóriaterületet kezelik. Az allokátorok lehetővé teszik, hogy egy konkrét környezet felállításához igazítsuk a memóriafoglalást, méghozzá úgy, hogy megőrizzük a konténeroszály interfészének hordozhatóságát.

Az allokátor-definíciókban megtalálhatjuk a *value_type*, *reference*, *size_type*, *pointer* és a *difference_type* típusokat. A legfontosabb allokátor-tagfüggvényeket az alábbi táblázatban foglaltuk össze:

Tagfüggvény	Leírás
<code>allocator()</code> ; <code>allocator(const allocator& a) ;</code> <code>~allocator()</code>	<code>pointer address (reference r) const</code> / címevel tér vissza, tárfoglalás <i>n</i> darab objektum számára, <code>void deallocate(pointer p, size_type n) ;</code> <code>size_type max_size () const ;</code> <code>a difference_type</code> legnagyobb értékével tér vissza.

Az alábbi példában saját allokátort készítettünk, és alkalmaztuk azt vektor létrehozása során.

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

```

template <class T> class myalloc {
public:
    typedef T value_type;
    typedef size_t size_type;
    typedef ptrdiff_t difference_type;
    typedef T* pointer;
    typedef const T* const_pointer;
    typedef T& reference;
    typedef const T& const_reference;

    void construct(pointer p, const_reference val) {
        new (p) T(val);
    }

    void destroy(pointer p) {
        p->~T();
    }

    pointer allocate(size_type sz, void* vp = 0) {
        pointer p = static_cast<pointer>(operator
            new(sz * sizeof(value_type)));
        cout << "allocate " << sz << " darab "
            << typeid(T).name() << " elem, a "
            << hex << (unsigned long) p << " címen" << endl;
        return p;
    }

    void deallocate(pointer p, size_type) {
        cout << "deallocate " << (unsigned long) p << endl;
        operator delete(p);
    }
};

void main()
{
    // alapelemezett alokator hasznalata
    vector<double> v(123, 12);
    // sajat alokator alkalmazasa
    vector<int>, myalloc<int> > myv(100, 23);
}

```

```

allocate 100 darab int elem, a 7f0280 címen
deallocate 7f0280

```

F5. Karaktertorsorozatok kezelése

A C++ nyelv könyvtárában a nullavégű karaktertorsorozatok kezelésére szolgáló C-függvényeket (*cstring*) és a sablonnal definiált sztringoszályokat (*string*) egyaránt megtaláljuk. A fejezetben először áttekinthetjük a C-könyvtár ide tartozó függvényeit, majd *basic_string<char>* sablonpéldánnyal foglalkozunk, melynek neve *string*.

F5.1 A C-könyvtár függvényei

A C nyelvben a karaktertorsorozat (*string*) '\0' karakterrel záródó karaktertömböt jelöl.

```
#include <iostream>
using namespace std;
int main(void)
{
    char s1[12], *s2;
    s2 = new char [12];
    strcpy(s1, "Lafentla");
    strcpy(s2, s1);
    cout<<s2<<" 4. karaktere: "<<s2[4]<<endl;
    cout<<"Lafentla"[4]<<endl;
    delete[] s2;
    cin.get();
}
```

Lafentla 4. karaktere: n

A sztringkezelő függvények prototípusát a *cstring* állomány tartalmazza. Ugyanígy találjuk a függvények széles karakterekkel (*wchar_t*) működő változatait. Példaként tekintsük az sztringmásoló függvény két változatának prototípusait!

```
char *strcpy(char *dest, const char *src);
```

```
wchar_t *wstrcpy(wchar_t *dest, const wchar_t *src);
```

Az alábbiakban csak a *char* típushoz készített függvényeket tekintjük át:

```
char *strcat(char *str1, const char *str2);
```

Az *strcat()* függvény az *str1* karaktertorsorozathoz fűzi az *str2* sztring tartalmát, és az összeűzött karaktertorsorozatra mutató pointerrel tér vissza.

```
char *strchr(const char *str1, int c);
```

Az *strchr()* függvény megkeresi a *c*-ben tárolt karakter első előfordulásának helyét az *str1* karaktertorsorozatban, és erre a pozícióra mutató pointerrel tér vissza. (A *NULL* függvényérték azt jelzi, hogy a karakter nincs a sztringben.)

```
int strcmp(const char *str1, const char *str2);

Az strcmp() függvény karakterenként összehasonlít két karaktersorozatot. A visszatérési érték negatív, ha  $str1 < str2$ , nulla, ha  $str1 == str2$  és pozitív, ha  $str1 > str2$ .

char *strcpy(char *str1, const char *str2);

Az strcpy() függvény az  $str2$  sztring karaktereit másolja az  $str1$  karaktersorozatba a záró EOS ( $\backslash 0$ ) karakterrel együtt.

size_t strlen(const char *str);

Az strlen() függvény a karaktersorozat hosszát adja vissza. A visszatérési érték a karakterek számát tartalmazza, a sztringet záró EOS karakter nélkül.

char *strncat(const char *str1, const char *str2, size_t maxn);

Az strncat() függvény az  $str2$  karaktersorozatból  $maxn$  karaktert hozzáfűz az  $str1$  sztring végéhez.

size_t strlen(const char *str);
```

Az *strlen()* függvény a karaktersorozat hosszát adja vissza. A visszatérési érték a karakterek számát tartalmazza, a sztringet záró EOS karakter nélkül.

Az *strncat()* függvény az *str2* karaktersorozatból *maxn* karaktert hozzáfűz az *str1* sztring végéhez.

```
int strcmp(const char *str1, const char *str2, size_t maxn);
```

Az *strcmp()* függvény az összehasonlításnál mindkét karaktersorozatból csak *maxn* számú karaktert vesz figyelembe. A visszatérési érték negatív, ha $str1 < str2$, nulla, ha $str1 == str2$ és pozitív, ha $str1 > str2$.

```
char *strncpy(const char *str1, const char *str2, size_t maxn);
```

Az *strncpy()* függvény az *str2* karaktersorozatból *maxn* karaktert másol az *str1* karaktersorozatba (a nulla karakter hozzáadása nélkül).

```
char *strncpy(const char *str1, const char *str2, size_t maxn);
```

Az *strncpy()* függvény az *str1* karaktersorozatban az *str2* sztring bármely karakterének első előfordulását keresi, és az előfordulási helyre mutató pointerrel tér vissza. Ha az *str2* semmilyen karaktere sem található meg az *str1*-ben, akkor *NULL* lesz a függvény értéke.

```
char *strchr(char *str, int c);
```

Az *strchr()* függvény az *str* sztringben a *c* karakter utolsó előfordulási helyét keresi, és az előfordulás helyére mutató pointerrel tér vissza. Ha a karakter nem található, akkor *NULL* lesz a függvény értéke.

```
size_t strspn(const char *s1, const char *s2);
```

Az *strspn()* függvény az *s1* sztring első olyan karakterének pozíciójával tér vissza, amely nem található meg az *s2* karakterisorozatban.

```
char *strstr(const char *str1, const char *str2);
```

Az *strstr()* függvény az *s2* részsorozat első előfordulási helyét keresi az *s1* karakterisorozatban. A visszatérési érték az első előfordulási helyre mutató pointer. Ha az *s2* nem található meg az *s1*-ben, *NULL* értékkel tér vissza a függvény.

```
char *strtok(char *str1, const char *str2);
```

A *strtok()* függvény az *s2* karakterisorozatban megadott elválasztójelek alapján – többszöri hívással – feldarabolja az *s1* sztringet. (A függvény második hívásától kezdődően az első argumentumban *NULL* értéket kell megadni.) Minden visszatérési érték egy-egy részsorozatra mutató pointer (vagy *NULL*, ha az eredeti sztring nem darabolható tovább). Az alábbi példában az *strtok()* függvény alkatmazásával szavakra bontjuk a megadott mondatot:

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

void main(void)
{
    char mondat[] = "A boldogság egy pillango; ha kergetjuk,"
        " kislálik a kezunkból... De ha csendben"
        " leülünk, az is lehet, hogy leszáll rank.";
    char *p;
    p = strtok(mondat, "rasjelek");
    do {
        cout<<p<<endl;
        p = strtok(NULL, "rasjelek");
    } while (p!=NULL);
    cin.get();
}
```

Az *strxfrm()* függvény átalakítja (transzformálja) az *s2* sztringet a helyi beállításoknak (LC_COLLATE) megfelelően, és a *s1* sztring első *maxn* karakterébe tölti az EOS karakterekkel együtt. Az

```
size_t strxfrm(char *str1, const char *str2, size_t maxn);
```

kifejezés megadja a célstring (*s1*) szükséges méretét. A "C" helyi beállítások esetén az *strxfrm()* függvény működése megegyezik az alábbi utasításokével:

```
strncpy(str1, str2, maxn);
return (strlen(str1));
```

```
int strcmp1(const char *str1, const char *str2);
```

Az *strcmp1()* függvény a helyi beállítások (*LC_COLLATE*) figyelembevételével összehasonlítja az *str1* és az *str2* sztringeket. A visszatérési érték negatív, ha *str1* < *str2*, nulla, ha *str1* = *str2* és pozitív, ha *str1* > *str2*.

F5.2 A string osztály használata

Mint a bevezetőben említettük, a *string* osztály a *basic_string* sablonosztály *char* típusú *wstring*.) A jobb érthetőség kedvéért, az osztálysablon elemei helyett, a megvalósított *string* osztály tagjait tekintjük át.

Konstruktorok

Egy sor konstruktor segíti a karaktertorsorozatok kényelmes definiálását:

Konstruktor	Leírás
<code>string()</code> ;	alapértelmezett, egy üres sztringet hoz létre,
<code>string(const char * p) ;</code>	konverziós konstruktor, amely c-karakter-sorozatból állítja elő a sztringet,
<code>string(const vector<char> &v) ;</code>	konverziós konstruktor, amely konténerből állítja elő a sztringet,
<code>string(const string & s, size_t pos=0, size_t n = npos) ;</code>	másoló konstruktor – <i>npos</i> értéke általában – <i>l</i> , ami azt jelzi, hogy nem volt memória foglaltva.
<code>string(const char * p, size_t n) ;</code>	konstruálás a <i>p</i> C-sztring első <i>n</i> karakteréből,
<code>string(const char c, size_t n=1) ;</code>	<i>n</i> darab <i>c</i> karakterből álló sztringet készít.

Tagfüggvények

A *string* osztály túlterhelt operátorai:

Operátorfüggvények	Leírás
<code>string& operator=(const string& s) ;</code>	értékkadás,
<code>string& operator=(const char* p) ;</code>	értékkadás,
<code>string& operator=(char c)</code>	értékkadás,
<code>string& operator+=(const string& s) ;</code>	hozzáírás,
<code>string& operator+=(const char* p) ;</code>	hozzáírás,
<code>string& operator+=(char c)</code>	hozzáírás,
<code>operator vector<char>() const ;</code>	vektorra konvertálás,
<code>char operator[] (size_t pos) const ;</code>	adott pozícióban lévő karakter,
<code>char& operator[] (size_t pos) ;</code>	adott pozícióban lévő karakter referenciája.

A további tagfüggvényeknek is több áttekinthető változata létezik a *string*, a *char ** és a *char* típusokhoz. A leírásban csupán a függvénynevre és a működés leírására szorítkozunk:

Tagfüggvények	Leírás
append()	hozzáfűzés,
assign()	értékdadás,
insert()	beszúrás adott pozíciótól,
remove()	adott pozíciótól kezdődően, adott számú karakter törlése,
replace()	adott számú karakter adott pozíciótól kezdődő felülírása,
get_at()	adott pozíción levő karakter lekérdezése,
put_at()	karakter betrása adott pozícióra,
length()	a karakter sorozat hosszának lekérdezése,
c_str()	konvertálás <i>char *</i> típusú karakter sorozattá,
data()	a karakter sorozat kezdőcímének lekérdezése,
resize()	a <i>string</i> átméretezése,
copy()	adott pozíciótól, adott számú karakter másolása <i>char *</i>
substr()	karakter sorozatba,
	adott pozíciótól, adott számú karakterből részsorozat
compare()	készítése,
	a tárolt <i>string</i> összehasonlítása egy másik karakter sorozattal,
find()	egy másik karakter/karakter sorozat keresése a tárolt
rfind()	sztringben,
	keresés visszafelé,
find_first_of()	első egyező előfordulás keresése,
find_last_of()	utolsó egyező előfordulás keresése,
find_first_not_of()	első nem egyező előfordulás keresése,
find_last_not_of()	utolsó nem egyező előfordulás keresése.

Külső operátorfüggvények

A *string* osztályhoz tartozik egy sor, külső függvényként megvalósított operátor is.

Tagfüggvények	Leírás
istream& operator>>(istream& is, string& str) :	adatbeviteli operátor,
istream& getline(istream& is, string& str, char delim) ;	adatbeviteli függvény,
ostream& operator<<(ostream& os, const string& str) ;	adatkiviteli operátor,
string operator + (const string& s1, const string& s2) ;	a megadott két <i>string</i>
bool operator==(const string& s1, const string& s2) ;	összehasonlítás,
	<i>true</i> , ha <i>s1</i> lexikografikusan
	azonos <i>s2</i> -vel.

```

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

void main() {
    string mondat, szavak[19];
    int poz=0, rpoz=0, nszo, i=0;

    mondat = " A boldogság egy pillango ha kergetjuk";
    mondat += " Kísiklik a kezunkbol de ha csendben"
    " Leulunk az is lehet hogy leszall rank";

    // Felbontjuk a mondatot szavakra a szóközöknél vágva
    while (poz<mondat.size()) {
        poz = mondat.find(' ', rpoz);
        szavak[i++] = assign(mondat, rpoz, poz-rpoz);
        cout<<szavak[i-1]<<endl;
        rpoz = poz+1;
    }
}

```

Végezetül nézzünk meg két példaprogramot az elmondottak szemléltetésére! Az elsőben szavakra bontjuk a megadott idézetet, míg a másodikban a valós számok formázott megjelenítését segítő függvényt mutatunk be.

```

bool operator==(const string& s, const char* cs);
bool operator==(const char* cs, const string& s);
bool operator==(const string& s1, const string& s2);

string operator+(const string& s, char c);
string operator+(const string& s, const char* cs);
string operator+(char c, const string& s);
string operator+(const char* cs, const string& s);
string operator+(const string& s1, const string& s2);

```

nek a különböző típusokhoz:

Az összehasonlító és az összeülő operátoroknak további túlterhelt változatai is létez-

```

bool operator!=(const string& s1, const string& s2);
bool operator<(const string& s1, const string& s2);
bool operator<=(const string& s1, const string& s2);
bool operator>(const string& s1, const string& s2);
bool operator>=(const string& s1, const string& s2);
bool operator<>(const string& s1, const string& s2);

true, ha s1 lexikografikusan
nem azonos s2-vel,
true, ha s1 lexikografikusan
kisebb s2-nél,
true, ha s1 lexikografikusan
nagyobb vagy egyenlő mint s2,
true, ha s1 lexikografikusan
nagyobb s2-nél,
true, ha s1 lexikografikusan
nagyobb vagy egyenlő s2.

```

```

nszo = 1; // a szavak száma
// osszerakjuk a mondatot szóhelyettesítéssel
mondat=" ";
for (i=0; i<nszo; i++)
    if (iszavak[i].compare("ha"))
        mondat+="amenyiben ";
    else
        mondat+=szavak[i]+' ';
mondat+="! ";
cout<<mondat<<endl;
cout<<"Helyes? [igen/nem] ";
getline(cin, mondat, '\n');
if (!mondat.compare("igen"))
    cout<<"Helyes!"<<endl;
    else
        cout<<"Nem tudom milyen."<<endl;
cin.get();
}

```

A formázott megjelenítés alapja, hogy a valós számot szöveggé alakítva egész- és törtészre bontjuk. A formázást végző függvény prototípusa:

```
string format(double d, int w=20, int p=7);
```

ahol a *d* a valós érték, *w* a formázott szám karaktereinek száma, *p* pedig a tizedesjegyek száma. Az eredménystringet függvényértékként kapjuk vissza. A *format()* függvény és a használatát bemutató példaprogram:

```

#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;

const int MAX = 12;

double adat1[MAX] =
    { 123.79, 123e-4, 12.2379, -0.1223,
      0.456, 3.14159265, -15232,
      34, 2331.01, -2323.42, 2e-3, -2.7342e-3 };

double sum(double d[], int n) {
    double s = 0;
    while (n-->-1) s += d[n];
    return s;
}

string format(double d, int w=20, int p=7) {
    // double -> string konverzió
    int egész, eljöl;
    string ds = fcvrt(d * 1E15, w+p, egész, eljöl);
    egész -= 15;
}

```

A program futásának eredménye:

+123.79	-250.84825402
+0.0123	-50.1696508
+12.2379	-10.03393016
-0.1223	-2.00678603
+0.456	-0.4013572
+3.14159	-0.08027144
*****	-0.01605428
+34.0	-0.00321085
+2331.01	-0.00064217
-2323.42	-0.00012843
+0.002	-0.00002568
-0.00273	-0.00000513

```

// az elöjellet tartalmazó sztring
string sg(1, elöjel ? '-' : '+');
// az egészrész-sztring előállítás
string se = ds.substr(0, egész);
if (egész>0)
    se = sg + se;
else
    se = sg + string("0");
int isp = w-p-l-se.length();
if (isp<0) return string(w, '*');
se = string(isp, ' ') + se + ".";
// a törtész-sztring előállítás
string st = (string(egész<0 ? -egész : 0, '0') +
ds.substr(egész<0 ? 0 : egész)).substr(0,p);
for (int i=st.length()-1; i>0 && st[i]!='0'; st[i]--) ;
return se+st;
}

void main() {
    double atl = sum(adatl, MAX) / MAX;
    double adat2[MAX];
    for (int k=0; k<MAX; k++)
        adat2[k]=(atl/=5);
    for (int k=0; k<MAX; k++) {
        cout<<format(adatl[k],11,5);
        cout<<format(adat2[k],18,8)<<endl;
    }
    cin.get();
}

```

Irodalomjegyzék

Benkő Tiborné - Poppe András
Együtt könnyebb a programozás
Objektum-orientált C++
ComputerBooks, Budapest, 2004

Nyisztor Károly

Gyakorlati C++

Kossuth Kiadó, Budapest, 2004

Bjarne Stroustrup

The C++ Programming Language

Special Edition

Addison-Wesley Publishing Co., 2000

Dirk Louis

C/C++, New Reference

Prentice Hall, 2001

Robert Robson

Using the STL

Springer-Verlag New York, Inc., 1988, 200

Victor Shtern

Core C++, A Software Engineer Approach

Prentice Hall Inc., 2000

Herbert Schildt

C/C++: Programmer's Reference

Second Edition

Osborne McGraw Hill, 2000

Johnsonbaugh, Richard

Applications Programming in C++

Prentice Hall Inc., 1999

- Herbert Schild*
Teach Yourself C++
Third Edition
 Osborne McGraw-Hill, 1998
- Ira Pohl*
Object-Oriented Programming Using C++
Second Edition
 Addison-Wesley Longman, 1997
- Timothy Budd*
Data Structure in C++ Using the STL
 Addison-Wesley Longman Inc., 1997
- Kris Jamsa*
C++
 Kossuth Kiadó, Budapest, 1997
- Ira Pohl*
C++ Distilled
 Addison-Wesley, 1997
- Dr. Kondorosi Károly - Dr. László Zoltán - Dr. Szirmai-Kalos László*
Objektum-orientált szoftverfejlesztés
C és C++ nyelven
 ComputerBooks, Budapest, 1997
- P. J. Plauger*
The Draft Standard C++ Library
 Prentice Hall, 1995
- Stanley B. Lippman*
C++ primer (2nd Edition)
 Addison-Wesley Publishing Co., 1991
- Brian W. Kernighan - Dennis M. Ritchie:*
The C programming language (second edition)
 Prentice Hall, Inc., New Jersey, 1988

- #
- #define, 27, 156, 181, 185
 - #elif, 183
 - #else, 183
 - #endif, 183
 - #error, 190
 - #if, 181, 183
 - #ifdef, 183
 - #ifndef, 183
 - #include, 17, 124, 182
 - #line, 190
 - #pragma, 116, 190
 - #undef, 185
-
- cplusplus, 189
 - DATE, 189
 - FILE, 189
 - LINE, 189
 - reverse_iterator, 312
 - STDC, 189
 - TIME, 189
- A
- abs, 299
 - Abstract Data Type, 194
 - absztrakt adattípus, 194, 197
 - absztrakt osztály, 237
 - accumulate, 313
 - acos, 296
 - adafolyam, 9, 267
 - adafolyam-iterátorok, 264
 - adatformázás, 265
 - adatrejés, 18, 193
 - adatsorkézeltés, 264, 305
 - adatsorok, 264, 305
 - adattag, 102, 197, 198
 - address, 317
- asszociatív tároló bitálmaz, 305
- asszociatív tárolók, 264, 305, 308
- ASSOC, 308
- assign, 323
- asset makrók, 263
- assert, 304
- asin, 296
- asctime, 300
- array, 83
- aritmetikai típusú paraméter, 130
- aritmetikai típusok, 21
- aritmetikai operátorok, 38
- argv, 149
- argumentumok kiértékelésének sorrendje, 127
- argumentumok, 123
- argc, 149
- append, 323
- app, 283
- ANSI C++ fordító, 189
- ANSI, 3, 11
- átalánosított osztály, 253
- átalánosított numerikus műveletek, 264
- átalánosított függvény, 159
- átalános szolgáltatások, 263
- átalános mutató, 30
- alprogram, 123
- állományok, 284
- allokátorok, 317
- allocators, 317
- allocator, 317
- allocate, 317
- algoritmus fejlődés, 101
- algoritmusok, 264, 313
- algoritmus, 264, 305
- alapítópók, 21
- alapoztály inicializálás, 235
- alapoztály, 194, 231
- alapértelmezés szerinti argumentum, 146
- aktuális sor száma, 189
- aktuális paraméterek, 123
- aktuális argumentumok, 123
- aggregate típusok, 21, 102
- ADT, 197
- adjacent_find, 313
- adjacent_difference, 314

- asszociativitás, 34, 36
- atan, 297
- atan2, 297
- arc, 283
- atoi, 292
- atoi, 292
- auto, 25, 86
- auto tárolási osztály, 175
- auto változók inicializálása, 176
- automatikus élettartam, 168
- automatikus konverziók, 55
- azonosítók, 11
- B**
- B nyelv, 2
- back, 310
- back_insert_iterator, 312
- backslash, 14
- bad_alloc kivétel, 52
- bad_cast kivétel, 57
- bad_typeid kivétel, 54
- bájt, 267
- balértek, 28
- barát, 202
- base, 194
- base class, 231
- Basic Combined Programming Language, 2
- basic_string<char>, 319
- BCPL, 2
- beágyazás, 234
- begin, 307
- beiső azonosító, 176
- beiső szintű deklaráció, 174
- bidirectional_iterator, 311
- bináris fájlok, 285
- binary, 287
- binary_search, 313
- binder1st, 316
- binder2nd, 316
- bitcitoló műveletek, 44
- bitenkénti logikai műveletek, 43
- bitimalmaz, 305
- bitmaszk, 43
- bitmező, 115
- bitset, 264, 305
- blokk, 60
- blokkszintű érvényességű tartomány, 168
- bool, 22
- boolalpba, 280
- break, 68, 73
- bsearch, 303
- C, CS**
- C nyelv, 2
- C#, 4
- c_str, 323
- C++ forráskódlomány, 189
- C++ nyelv bővítése, 227
- callloc, 295
- case, 65
- cassert, 263
- cast, 30, 55
- catch, 77
- CC, 307
- cctype, 264, 265, 291
- ceil, 299
- cerr, 10, 267, 277
- cerrno, 263, 265
- cfloat, 263, 265
- char, 267
- ciklusutasítások, 67
- cim szerinti argumentum átadás, 128
- cime operátor, 47, 88, 105
- cin, 9, 90, 267, 277
- cin.getline, 90
- ciso646, 265
- class, 21, 23, 111, 193, 197
- class generator, 253
- climits, 263, 265
- locale, 264, 265
- clock, 300
- clog, 267, 277
- close, 288
- cmath, 264, 265, 296
- compare, 323
- compiling, 18
- complex, 264
- const, 23, 26, 57, 181, 200
- const_reverse_iterator, 306
- const_cast operátor, 56
- const_iterator, 306
- const_reference, 306
- container_type, 306
- continue, 68, 74
- copy, 314, 323
- copy konstruktor, 205, 210
- copy_backward, 314
- cos, 297

cosh, 297
 count, 309
 count_if, 313
 count_min, 313
 cout, 9, 90, 267, 277
 CR/LF, 285
 ctime, 263, 265, 300, 301
 ctype, 265
 cwchar, 264, 265
 cwstring, 264, 265
 csetjmp, 263, 265
 csignal, 263, 265
 csidarg, 148, 263, 265
 csiddef, 263, 265
 csidio, 265, 267
 csidlib, 149, 263, 264, 265, 292, 295
 cstring, 90, 264, 265, 294, 319

D

data, 323
 data hiding, 193
 datamember, 102, 197
 dátum- és időkezelő függvények, 300
 dátumkezelés, 263
 deallocate, 317
 debug, 183
 dec, 281
 decimalis szám, 13
 decrement, 41
 default, 65
 default argumentum, 146
 default konstruktor, 210, 211
 defined, 183
 definició, 21, 81, 123
 deklaráció, 21, 124
 deklarációs állományok, 17
 delete, 204, 215
 delete operátor, 51
 delete töltéshélese, 226
 delete[], 52, 97, 109
 deque, 264, 305, 309
 derived, 21, 194
 derived class, 231
 destructor, 194, 209, 215
 difference_type, 306, 317
 difftime, 301
 dinamikus elletartam, 168
 dinamikus helyfoglalástómb, 97
 dinamikus memóriafoglalás, 95
 cosh, 297
 count, 309
 count_if, 313
 count_min, 313
 cout, 9, 90, 267, 277
 CR/LF, 285
 ctime, 263, 265, 300, 301
 ctype, 265
 cwchar, 264, 265
 cwstring, 264, 265
 csetjmp, 263, 265
 csignal, 263, 265
 csidarg, 148, 263, 265
 csiddef, 263, 265
 csidio, 265, 267
 csidlib, 149, 263, 264, 265, 292, 295
 cstring, 90, 264, 265, 294, 319

E, F

dinamikus memóriakezelés, 263, 295
 dinamikus tömb, 51, 305
 dinamikus típusátalakítás, 57
 dinamikus összerendezés, 237
 direktíva, 181
 div, 299
 divides, 315
 double, 15
 do-while ciklus, 71
 dynamic_cast, 57, 241
 early binding, 195
 eavl, 294
 EDOM, 296
 egész konstans, 13
 egész konverzió, 55
 egész típusok, 21
 egydimenziós tömb, 83
 egyoperandusú operátorok, 33
 egyszerű indirektitű mutató, 87
 egyszerű változók, 25
 elérési mód, 288
 élettartam, 167, 174
 elfedés, 175
 előfeldolgozó, 181
 előfordító, 12
 előjel nélküli egész, 13
 előkészítő ciklus, 67
 előrevetített deklarációk, 265
 előrevetített operátor, 33
 else, 62
 else-if szerkezet, 63
 elsőbbségi szabály, 36
 elsődleges kifejezések, 33
 eltolás balra, 44
 eltolás jobbra, 44
 empty, 307, 310
 encapsulation, 193
 end, 307
 endl, 280
 ends, 280
 enum, 13, 23, 24, 101, 170
 EOF, 284
 EOLN, 284
 equal, 313
 equal_range, 309, 313
 equal_to, 315
 ERANGE, 296

getws, 268
globális élettartam, 167
globális névter, 170
gmtime, 301
GNU, 2
goto, 59, 76
greater, 315
greater_equal, 315

H

handler, 78
Hanoi tornyai, 152
háromoperandusú operátor, 33
hatékony feloldó operátor, 38
hatékony operátor, 171
hátírvetett operátor, 33
háluitesztelő ciklus, 67
hatvány- és logaritmusfüggvények, 298
header file, 182
Hewlett-Packard Company, 305
hex, 281
hexadecimalis konstans, 13
hibakeresés, 183
hibakezelés, 263
hibakódok, 263
hibátűrő programok, 77
hivalkozás szerinti paraméter-átadás, 129
hivalkozási típus, 29
hordozhatóság, 3
hosszi egész, 13
HUGE_VAL, 296
hullám karakter, 215

I

I/O művelet, 267
időkezelés, 263
if, 50, 60
if-else szerkezet, 62
ifstream, 287
ignore, 279, 289
implementáció, 263
implicit konverziók, 55
implicit mutatókonverzió, 56
implicit példányosítás, 253
implicit típuskonverziók, 55
includes, 313
increment, 41

G

friend deklaráció, 203
friend mechanizmus, 203
friend viszony, 233
from, 310
from_insert_iterator, 312
fscanf, 268, 286
fsseek, 286
fstream, 265, 287
ftell, 286
function adaptors, 316
functional, 263, 305, 315
functionoidok, 223
futas közbeni típusinformációk, 241
futas közötti típusinformációk, 241
futásidőjelelő típusazonosítás, 54, 57
futtatható program, 18
futtatható program, 18
függvény, 123
függvény szintű erőnyességi tartomány, 169
függvényargumentum, 138
függvényattribútum, 127
függvényérték, 126
függvényhívás, 127
függvényminia, 159
függvénynév áidefiníálása, 157
függvényre mutató pointer, 139
függvényre mutató pointerek tömbje, 142
függvényre mutató pointer, 157
függvényiszertű makró, 186
függvényiszertű osztályok, 223
fwrite, 268
fwrite, 268, 286
fwscanf, 268

- külső szintű deklaráció, 174
- L**
- labs, 299
- late binding, 195
- láthatóság, 18, 168, 174
- ldexp, 298
- ldiv, 299
- lebegőpontos konstans, 15
- left, 281
- leképzések, 305
- length, 323
- leptető operátorok, 41
- leptető operátorok túlterhelése, 220
- less, 315
- less_equal, 315
- lesztírozott osztály, 194
- lexicographical_compare, 313
- lifetime, 167
- limits, 263, 265
- lineáris lista, 118, 305
- linkage, 169
- linking, 18
- list, 264, 305, 309
- listelem ürítése, 120
- listaszerkezet, 118
- literal, 15
- local, 264, 265
- localtime, 301
- log, 298
- log10, 298
- logical_and, 315
- logical_not, 315
- logical_or, 315
- logikai ES, 40
- logikai operátorok, 40
- logikai VAGY, 40
- lokális élettartam, 168
- long, 13
- long double, 15
- loop, 67
- lower_bound, 309, 313
- lua, 293
- lua, 28
- M**
- main, 4, 17, 149
- műveleti elemek, 263, 305
- műveletadapterek, 316
- mutatóömb, 94
- mutatóoperátor, 47
- mutató, 28, 87
- mutable típusminősítő, 200
- mutable, 201
- multplies, 315
- multiple inheritance, 194, 231
- modulus, 315
- moduláris programozás, 18, 177
- modf, 299
- mtime, 301
- mismatch, 313
- minus, 315
- minimális megszűrésesség, 270
- min_element, 313
- min, 313
- metódus, 193
- merge, 314
- mértimódosító előtag, 272
- memset, 295
- memory, 263, 305, 317
- memóriakezelés, 263, 305
- memmove, 295
- memcpy, 294
- memcmp, 294
- memchr, 294
- memberfunction, 197
- melikékhatis, 33, 37
- megszakításos modell, 77
- megosztási mód, 288
- megjegyzés, 16
- max_size, 307, 317
- max_element, 313
- max, 313
- math, 265
- matematikai függvények, 296
- maszkolás, 43
- másoló konstruktor, 205, 210
- map, 264, 305
- manipulátorok, 265, 279
- malloc, 51, 295
- makróoperátor, 188
- makrókifejtés, 185
- makróhívás, 186
- makróhelyettesítés, 185
- makró, 12, 185
- make_heap, 314
- main() függvény visszatérési értéke, 4, 149
- main() függvény paraméterei, 149

műveletobjektumok, 263, 305, 315

N,NY

namespace, 18, 124, 170, 171

negate, 315

névelen névelület, 169, 177

névelen union, 114

névelület, 18, 124, 170

new, 97, 109, 209, 215, 263

new operator, 51

new túlterhelése, 226

next_permutation, 314

NL, 285

noboolealpha, 280

noshowbase, 280

noshowpoint, 280

noshowpos, 280

noskipws, 280

not_equal_to, 315

not1, 316

not2, 316

nothrow memóriafoglaló, 52

noubuf, 280

noupptcscase, 280

nth_element, 313, 314

NUL, 13, 119, 134, 150, 215

nullavégtű karakterláncok, 264

numeric, 264, 305

numerikus könyvtár, 264

numerikus műveletek, 305

nyelvi támogatás, 263

nyilvános, 197

nyomlatítható karakterek, 291

O

object modul, 18

objektumok hierarchiája, 193

oct, 281

ofstream, 287

oktális konstans, 13

olvasás sztringből, 283

open, 287

operandusok, 16, 33

operator átdefiniálása, 194

operator függvény, 315

operator overloading, 36, 106, 158, 194, 217

operator túlterhelés, 194

paraméter nélküli konstruktor, 210
 paraméter típus, 126
 paraméter-deklarációs lista, 124
 paraméterek, 123
 paraméterezett osztály, 253
 partial_sort, 314
 partial_sort_copy, 314
 partial_sum, 314
 partition, 314
 peek, 289
 példány inicializálás, 209
 példány megszüntetés, 209
 példányosítás, 253
 Perl, 5
 PHP, 5
 plus, 315

P

önrekurzív, 153
 öröklés, 194, 231
 öröklődés, 231
 öröklött tagok elérése, 233
 osztály, 194
 összehasonlító operátorok, 39
 összetett értékadás, 46
 összetett utasítás, 60

Q

operator[], 322
 operátorfüggvény, 217
 operátorok, 11, 16, 33
 operátorok átdefiniálása, 36
 operátorok túlterhelése, 217
 országfüggető beállítások, 264
 ostream, 225, 265, 277, 287
 ostream_iterator, 311
 ostringstream, 283
 osztály, 193
 osztály példány, 204
 osztályfej, 197
 osztálynév, 197
 osztálypéldány, 193
 osztályszintű érvényességi tartomány, 169
 osztálytagra mutató pointer, 207
 osztálytípus, 111
 osztálytípus, 197
 osztálytípus, 197
 overload, 157, 199

R

rand, 73, 300
 random_access_iterator, 311
 random_shuffle, 314
 rbegin, 307
 read, 289
 realloc, 295
 redefine, 195, 237
 reference, 306, 317
 referencia, 29
 register, 25, 124
 register tárolási osztály, 180
 reinterpret_cast, 57
 rejteit osztály, 212
 rekurzív deklaráció, 108
 rekurzív függvény, 151
 remove, 314, 323
 remove_copy, 314
 remove_copy_if, 314
 remove_if, 314
 rend, 307
 rendezés, 264, 303
 replace, 314, 323
 replace_copy, 314
 replace_copy_if, 314
 replace_if, 314
 resetioflags, 281
 return, 66, 68, 77, 123
 reverse, 314
 reverse_copy, 314
 reverse_iterator, 306, 312
 rezise, 323
 rfind, 323
 right, 281
 rotate, 314
 rotate_copy, 314
 rövidzár, 38
 RTTI, 57, 241
 run time type identification, 57
 RunTime Type Information, 54, 241
 rvalue, 28
 sablon, 250
 sablonhierarchia, 257
 sablonosztály, 253
 sablonosztály „barátai”, 255

S,SZ**Ö**

pointer, 306, 317
 pointer-aritmetika, 39
 pointerműveletek, 47
 polimorfizmus, 237
 polimorfizmus, 195
 pop, 310
 pop_heap, 314
 portabilitás, 3
 postfix, 33
 pow, 298
 pozícionálás, 289
 precedencia, 34, 36
 prefix, 33
 preproceszor, 181
 preproceszorparancs, 181
 prev_permutation, 314
 printf, 9, 90, 147, 268, 269
 prioritásos sor, 305
 private, 112, 193, 197, 202, 233
 programbefejezés, 263
 programgenerátor, 190
 programindítás, 263
 projekt, 19
 protected, 112, 197, 202, 233
 prototípus, 18, 123
 prototípuszintű hatáskör, 169
 ptr_fun, 316
 ptrdiff_t, 312
 public, 112, 193, 197, 202, 233
 pure virtual function, 237
 push, 310
 push_heap, 314
 put, 279, 290
 put_at, 323
 putback, 289
 putc, 268
 putchar, 268, 269
 puts, 90, 268, 269
 putw, 268
 putwc, 268
 putchar, 268
 putws, 268
 Python, 5
 qsort, 140, 141, 303
 queue, 264, 305

- sablonosztály statikus adattagjai, 255
 saját, 197
 saját manipulátor, 282
 saját névterület, 170
 saját típus, 26
 scanf, 9, 90, 268, 273
 scientific, 281
 scope, 18, 168
 search, 313
 search_n, 313
 seekg, 289
 seekp, 290
 SEQ, 307
 set, 264, 305
 set_difference, 314
 set_intersection, 314
 set_new_handler, 53
 set_symmetric_difference, 314
 set_terminate, 79
 set_unexpected, 79
 set_union, 314
 setbase, 281
 setf, 279
 setfill, 281
 setiosflags, 281
 setjmp, 265
 setlocale, 304
 setprecision, 281
 setw, 281
 shift, 44
 short-circuit, 38
 showbase, 280
 showpoint, 280
 showpos, 280
 side effect, 33, 37
 signal, 265
 Simula 67, 3
 sin, 297
 sinh, 297
 size, 307, 310
 size_type, 306, 317
 sizeof, 48, 84
 skalar típusok, 21
 skipws, 280
 sor, 305
 soros konténerek, 307
 sort, 101, 314
 sort_heap, 314
 specializáció, 253
 sprintf, 268, 276
 sqrt, 298
 srand, 300
 random, 179
 sscanf, 268, 276
 stringstream, 265, 283
 stable_partition, 314
 stable_sort, 314
 stack, 255, 264, 305, 309
 Standard Template Library, 1, 263, 305
 static, 25, 86, 124, 169, 206
 static tárolási osztály, 177
 static_cast, 56, 57
 statikus belső szintű változók, 179
 statikus elemtartam, 167
 statikus osztálytagok, 206
 statikus típusátalakítások, 56, 57
 std, 10
 std névterület, 265
 sidarg, 265
 siddef, 265
 siderr, 267
 sidexcept, 263
 sidin, 267
 sidio, 265
 sidio adatfolyamok átírányítása, 276
 sidlib, 265
 sidout, 267
 sidout adatfolyamok átírányítása, 276
 STL, 91, 263, 305
 stercat, 90, 319
 sterr, 319
 stercmp, 90, 320
 strcpy, 90, 136, 319, 320
 stercspn, 320
 stream, 9, 267
 streambuf, 265, 283
 string, 264, 265, 319, 322
 string fejtáblomány, 91
 string osztály, 322
 string típus, 91
 string&, 322
 stlen, 90, 320
 stncat, 320
 stncmp, 320
 stncpy, 320
 stpbrk, 320
 sttrchr, 320
 sttrspn, 321
 sttrstr, 321
 stringstream, 283
 strtod, 293
 strtok, 321

terminate, 79
 termination, 77
 this, 38, 200, 205
 throw, 77
 tilde, 215
 time, 265, 301
 típusazonosság, 263
 típuskonverzió, 30, 55
 típuskonverziós operátor, 55
 típusmegőrző szerkesztés, 161
 típusminősítők, 23
 típusmódosítók, 23
 típusnév, 170
 típusok, 263
 típusok hierarchiája szerinti konverzió, 56
 tisztán virtuális függvény, 237
 tm, 300
 token, 11
 tolower, 292
 top, 310
 toupper, 292
 többalakúság, 195
 többdimenziós tömb, 92
 többszörös értékadás, 46
 többszörös indirektiségű mutató, 30
 többszörös öröklődés, 194, 231
 tömb, 83
 tömb átadása függvénynek, 133
 tömb inicializálása, 85
 transform, 314
 translation, 18
 trigonometrikus függvények, 296
 true, 13, 39
 try-blokk, 78
 típusordulás, 296
 type_info, 54
 typedef, 26, 86, 101, 138, 174, 207, 256
 type_info, 54
 type-safe linking, 161
 typename kulcsszó, 256
 typeinfo, 263
 typeid, 54, 241
 ungetc, 268
 ungetwc, 268
 unicode, 11, 14
 unary, 33
 unexpected, 79
 union, 293

U

strtoul, 293
 struct, 21, 23, 102, 197
 struktúraelem, 102
 struktúratömb, 108
 substr, 323
 Sun, 4
 swap, 307, 314
 swap_ranges, 314
 switch utasítás, 65
 swprintf, 268
 swscanf, 268
 system, 304
 szabványos sablonkönyvtár, 1, 91, 305
 számtömbök, 264
 származtatott adat típusok, 83
 származtatott osztályok, 231
 származtatott típusok, 21
 szekvenciális állományok, 284
 szimbolikus konstans, 185, 186
 szokásos aritmetikai konverziók, 56
 szöveges állományok, 284
 string, 89, 264
 string végét jelző bajt, 89
 stringadatfolyamok, 265
 stringargumentum, 136
 stringkonstans, 15, 89
 stringliterálok, 11, 89
 stringmásolás, 136
 strtointb, 94
 szűrés, 275

T,TV

tagfüggvények, 197, 198
 tagfüggvények újradefiniálása, 195
 taginicializációs lista, 213, 235
 tagosztályok inicializálása, 213
 tanh, 297
 tárgymodul, 18
 tárolási osztály, 124, 167
 auto, 86
 extern, 18, 86
 static, 86
 tárolók, 264, 305
 tellg, 289
 tellp, 290
 template, 158, 159, 250
 template<class >, 159

unicode sztringkonstans, 15
 union, 21, 23, 112
 unique, 314
 unique_copy, 314
 unibuf, 280
 UNIX, 2
 unset, 279
 unsigned, 13
 upper_bound, 309
 uppercase, 280
 using, 172
 using namespace, 51, 172
 utasításcímke, 170
 utility, 263, 305
 U
 üres direktíva, 190
 üres típus, 22
 üres utasítás, 59
 V
 va_arg, 148
 va_end, 148
 va_start, 148
 valarray, 264
 változó hosszúságú argumentumlista, 147
 value_compare, 308
 value_type, 306, 317
 variáns rekord, 113
 vector, 264, 305, 309
 vector<char>, 322
 védett, 197
 végtelen ciklus, 67
 vektorargumentum, 133
 véletlenszám-generátor, 73
 verem, 255, 305
 vessző operátor, 49
 vezérlőkarakterek, 291
 vfprintf, 268

wcerr, 10, 267
 wchar, 265
 wchar_t, 22, 267
 wcin, 10, 267
 wcout, 10, 267
 wcopy, 319
 wctod, 293
 wctol, 293
 while ciklus, 68
 whitespace, 11, 273
 write, 290
 ws, 280
 wscanf, 268

W

wscanf, 268
 wprintf, 268
 VT, 239
 vscanf, 268
 vsprintf, 268
 vsscanf, 268
 vsprintf, 268
 vscanf, 268
 vprintf, 268
 volatile, 23
 void típusú függvény, 77
 void, 22, 83, 126
 visszatérési érték típusa, 124
 visszaadott érték, 126
 visibilitás, 168
 virtuális tagfüggvény, 195, 237
 virtuális tábla, 239
 virtuális függvények újrafelfűzése, 237
 virtuális destruktorkok, 239
 virtuális alapszálak, 240
 virtual, 195, 216, 237, 241
 vfwscanf, 268
 wscanf, 268

3.497.-	Adatkezelés az MS ACCESS 2000 alkalmazásával	... Kovács L.: Adatbázisok tervezésének és kezelésének módszertana
4.200.-		... Abonyi J. és társai. Adatbányászat a hatékonyság eszköze
4.900.-		... Czenky M. Adatmodellezés * SQL és Access alkalmazás *
4.552.-	SQL Server ADO	... Dr. Hatvány Béla Csaba: ASP.NET vezérlők programozása
5.990.-		... Pintér M.: AutoCAD tankönyv és példatár 2006 verzióhoz
4.995.-		... Mórész Attila: Tippek a CD íráshoz
2.500.-		... Kuzmina J.-Dr. Tamás P.-Tóth B.: Windows alkalmazások fejlesztése
3900.-	C++Builder 6 rendszerten (e-könyv CD-n)	... Benkő L.-Benkő Tné-Tóth B.: Programozunk C nyelven
2.968.-		... Benkő Tné-Poppe A.: Együtt könnyebb a programozás - C -
3.220.-		... Benkő Tné-Tóth B.: Együtt könnyebb a programozás JAVA
3.990.-		... Tóth B.: Programozunk C++ nyelven !
2.994.-		... Benkő Tné-Poppe A.: Együtt könnyebb a programozás -
3.217.-	Objektum-orientált C++	... Köhalmi Éva-Köhalmi Martann: CorelDraw X3
7.900.-		... Laszlo József: Dinamikus weboldalak programozása -
3.150.-		... Kovalcsik Géza: Az EXCEL programozása
4.800.-		... Kovalcsikné Pintér Orsolya: Az EXCEL függvényei A-tól Z-ig
3.779.-		... Léva László: Excel 2003 táblázatkezelés és programozás a gyakorlatban 120 feladattal
2.600.-		... Laszlo József: Hangkártya programozása Pascal és Assembly nyelven
1.900.-		... Szirmai-Kalos: Hatromdimenziós grafika, animáció és játékefejlesztés
5.496.-		... Laszlo József: Internet a Világhálózat Net, Web meg a többiek
2.900.-		... Molnár Mátyás: Lotus Notes 6 felhasználóknak - e-könyv
2.915.-		... Dr. Kovácsné C.J.-Ozsváth M.-G.Nagy J.: Mit kell tudni Az EDC vizsgához?
2.995.-		... Dr. Kovács T.-Dr. Kovácsné C.J.-Ozsváth M.-G.Nagy J.: Mit kell tudni? a PC-ről -
1.999.-		Operációs rendszer, Szövegszerkesztés, Táblázatkezelés, Prezentáció, Internet és levelezés
990.-	Mobiltelefon a kezekben	... Mórész A.: Programozási alapfeladatok - WAP oldalakhoz és WEB lapokhoz
2.093.-		... Dr. Kovácsné Cohnér J.-Ozsváth M.-G. Nagy J.: OFFICE 2000
3.398.-		... Tóth B.-Benkő Tné és társai: Programozunk Turbo Pascal nyelven
2.992.-		... Tóth B.-Benkő Tné és társai: Együtt könnyebb a programozás Free Pascal
3.300.-		... Tamás P.-Tóth B. és társai: Programozunk DELPHI 7 rendszerben
3.860.-		... Köhalmi Éva-Köhalmi Tünde: Photoshop CS2 alapok és trükkök
6.450.-		... Laszlo József: PC hardver programozás - valós és védett módban
4.720.-		

Megrendelő neve	Számlázási cím	Megjegyzés
... Tátrai T.: MS Project 2000		
... Mudri István dr.: QuarkXpress 5		
... Dr. Hatványi Béla Csaba: Bit képek feldolgozása VISUL BASIC		
rendszerekben - © - (feladatok C++ és Delphi nyelven)		
... Tamas P. - Tóth B. és társai.: Programozunk Visual Basic rendszerben - ©		
... Nagy S.: Útleve a digitális világhoz - Nyilvános kulcsú tanúsítványok a gyakorlatban		
Kriptográfia, digitális aláírások, Hitelesítés szolgáltatás, Webáruházak		
biztonsága, Elektronikus bank biztonság, Tanúsítványok használata		
... Nagy S.: Elektronikus leveleink védelme		
Titkosítás, Digitális aláírás, Szabványos tanúsítványok		
... Gondank Tibor: Szemanitikus WEB Bevezetés a tudásalapú Internet világába		
... Gondank Tibor: WEBszolgáltatások - ©		
... Gerő Judit: WORD 2000 I. kötet - kezdőknek és haladóknak		
... Gerő Judit: WORD 2000 II. kötet - haladóknak		
... Gerő J.-Reich G.: WORD for WINDOWS 6.0 - magyar & angol		
... Gerő J.-Reich G.: WORD for WINDOWS 6.0 - magyar & angol		
Menedzsment informaticai szakkönyvek:		
... Szerk.: Heteyi J.: ERP rendszerek Magyarországon a 21. században		
... Szerk.: Heteyi J.: Vezetői döntéstámogató és elektronikus kereskedelmi		
rendszerek Magyarországon		
... Szerk.: Heteyi J.: Pénzüntézetek és állami intézmények információs rendszerei		
Magyarországon		
Gazdasági informatica:		
... Levaýné Lakner Mária: EXCEL táblázatkezelő a gyakorlatban -		
... Balogh Gábor: Visual Basic és EXCEL programozás -		
A Weboldalunkon részletes ismertetést talál könyveinkről:		
www.computerbooks.hu		