Objektum-orientált programozás C++ nyelven

ComputerBooks

BENKŐ TIBORNÉ
BENKŐ LÁSZLÓ
POPPE ANDRÁS

Objektum-orientált programozás C++ nyelven

C++ PROGRAM LÉPÉSRŐL-LÉPÉSRE
A NYELV TULAJDONSÁGAI
C++ MINT OBJEKTUM-ORIENTÁLT NYELV
TÖBBSZINTŰ ÖRÖKLŐDÉS
VIRTUÁLIS TAGFÜGGVÉNYEK
OPERÁTOROK ÁTDEFINIÁLÁSA
GRAFIKA, ANIMÁCIÓ

LEKTOR
SZŰCS GÁBOR

COMPUTERBOOKS
BUDAPEST, 1998

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

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

© Benkő Tiborné, Benkő László, Dr.Poppe András, 1997

©Kiadó: ComputerBooks Kiadói, Szolgáltató és Kereskedő Kft.
1126 Bp., Tartsay Vilmos u. 12.
Telefon: 175-15-64; Tel/Fax: 175-35-91
e-mail: info@computerbooks.hu
Felelős kiadó: ComputerBooks Kft ügyvezetője
ISBN: 963 618 157 8

Borítóterv: Székely Edith

Nyomtatta és kötötte a Dabas-Jegyzet Kft.
Felelős vezető: Marosi György ügyvezető igazgató
Munkaszám: 97-0618

Tartalomjegyzék

Bevezetés 5

1. C++ program lépésről-lépésre 9

1.1. Az első C++ program 9
1.2. Megjegyzések 10
1.3. A C++ nyelv kulcsszavai 11
1.4. A C++ nyelv azonosítói 11
1.5. Alaptípusok 12
1.6. Konstansok 12
1.6.1. Egész konstansok 12
1.6.2. Lebegőpontos konstansok 14
1.6.3. A 32 bites adatok típusai, méretei és határai 14
1.6.4. A karakter konstansok 14
1.6.5. Sztring konstansok 16
1.6.6. Enum konstansok 16
1.7. Tárolási osztályok, hatáskörök 16
1.7.1. Függvények tárolási osztályai 17
1.7.2. Változók élettartama és hatásköre 17
1.7.3. Egyszerű deklarátorok 19
1.7.4. Módosító jelzők 22
1.7.5. Típusdefiniáló (typedef) azonosítók 24
1.8. A szabványos Input/Output használata 25
1.8.1. Szabványos output 30
1.8.2. Szabványos input 32
1.8.3. A hibajelzés használata 33
1.8.4. A szabványos I/O bemutatása példákon keresztül 33
1.8.5. Az unset használata 38
1.8.6. A get és a put 38
1.8.6.1. Példaprogramok a get és a put használatára 38
1.9. Kifejezések 40
1.9.1. Elsődleges kifejezések 40
1.9.2. Operátorok 41
1.9.2.1. Egyoperandusú operátorok 42
1.9.2.2. Kétoperandusú operátorok 44
1.10. Konverziók 50

1.10.1. Aritmetikai konverziók 50
1.10.2. Konverzió a char, az int és az enum típusok között 51
1.11. Utasítások 52
1.11.1. Kifejezés-utasítások 52
1.11.2. Feltételes utasítás 54
1.11.3. Egyéb vezérlésátadó utasítások 58
1.11.4. Ciklusutasítások 59
1.12. Mutatók 63
1.12.1. A mutatók használata 63
1.12.2. A void * típusú mutatók 64
1.12.3. Mutatók értékadása 64
1.13. Tömbök 67
1.13.1. Kapcsolat tömbök és mutatók között 69
1.14. Struktúrák és unionok 72
1.14.1. Struktúrák megadása 72
1.14.2. Hivatkozás a struktúra elemekre 73
1.14.3. A bitmezők 77
1.14.4. A union fogalma 79
1.15. Függvények 81
1.15.1. Előredefiniált függvények használata 83
1.15.2. Típusváltoztató (type cast) operátor 84
1.15.3. Programozó által definiált függvények 85
1.16. Fájl kezelése 93
1.16.1 Karakter írása és olvasása 93
1.16.2. Különféle típusú adatok fájlkezelése 96
1.16.3. String fájlkezelése 101
1.16.4. Az ios osztály használata 104
1.16.5. Bináris filekezelés 106
1.17. A main függvény 111
1.18. Az előfeldolgozó 114
1.18.1. Szimbólumok és makrók 114
1.18.2. Feltételes fordítás 118
1.18.3. Előredefiniált szabványos szimbólumok 119
1.18.4. A Borland C++ saját előredefiniált szimbólumai az 5.02 verzióban 119
1.18.5. Fájlbeépítés 120
1.18.6. Fordítási hibaüzenetek 121
1.18.7. Implementáció-függő vezérlősorok 121

2. C++ nyelv tulajdonságai 125
2.1. C++ hivatkozás (referencia) használata 125
2.1.1. Egyszerű hivatkozások 125
2.1.2. Hivatkozási típusú függvényparaméter 126
2.2. C++ operátorok 128
2.2.1. Az érvényességikör operátor :: 128
2.2.2. A new és a delete operátorok 129

3. A C++ mint objektum-orientált nyelv 135
3.1. Az OOP alapjai 135
3.2. Egységbezárás 136
3.2.1. Alapfeladat: Műveletvégzés struct használatával 136
3.2.2. A Műveletvégzés feladat objektum-orientált változatai 138
3.2.2.1. Statikus helyfoglalású objektumpéldány 139
3.2.2.2. Tagfüggvények inline definíciója 143
3.2.2.3. Statikus helyfoglalású objektumpéldányok 144
3.2.2.4. Az adattagok privát (private) elérése 145
3.2.2.5. Az adattagok protected elérése 146
3.2.2.6. A new és a delete operátorok használata 147
3.2.2.7. A konstruktor 148
3.2.2.8. A destruktor 149
3.2.2.9. Konstruktor használata statikus objektum esetén 150
3.2.2.10. Paraméterezett típusok (templates) 152
3.2.2.11. Dinamikus helyfoglalású objektumpéldány 158
3.2.2.12. Dinamikus helyfoglalású objektumpéldányok 160
3.2.2.13. Objektum adattagjainak dinamikus létrehozása new eljárással 162
3.3. Öröklés 165
3.3.1. Öröklés az objektum-hierarchiában 166
3.3.2. Objektum öröklése 169
3.3.3. Objektum öröklése a Szamol tagfüggvény újradefiniálásával...................... 172
3.3.4. Az objektumok zártsága 175
3.4. Sokalakúság (polimorfizmus) 178
3.4.1. Virtuális tagfüggvények 178
3.4.2. Osztály típusú adattagra mutató pointer 182
3.4.3. A sablonok alkalmazása öröklött objektumok esetén 185
3. 5. További lehetőségek a C++-ban 188
3.5.1. Rokonok és barátok 188

3.5.2. Függvények értelmezésének kiterjesztése (*function overloading*) 190
3.5.3. Operátorok értelmezésének kiterjesztése (*operator overloading*) 192
3.5.4. Operátor-függvények definiálása 195
3.6. Mintapéldák műveleti jelek értelmezésének kiterjesztésére 198
3.6.1. A halmaz típus műveletei 198
3.6.2. A sztring típus és a hozzátartozó műveletek definiálása 213
3.7. Nagyobb méretű programok futtatása 233
3.7.1. Project fájl betöltése Borland 3.1 verzió esetén 233
3.7.2. Project fájl létrehozása 233
3.7.3. Műveletvégző feladat több modulban 234
3.7.4. Futtatás Borland C++ 5.02 rendszerben 237

4. C++ programok szöveges és grafikus üzemmódban 239
4.1. Szöveg és grafika 239
4.1.1. Programozás szöveges üzemmódban 240
4.1.2. Programozás grafikus üzemmódban 245
4.2. Grafikus programok és animáció 252
4.2.1. Halászás 252
4.2.2. Fénysorompó animációja 259

F1. Új nyelvi elemek a Borland C++ 5.02 implementációban 265
F1.1. Logikai típus 265
F1.2. Új konverziós operátorok 265
F1.3. Futásidejű típusazonosítás 267
F1.4. A typename kulcsszó 267
F1.5. Kivételek kezelése 268
F1.6. Nevek érvényességének korlátozása 269
F1.7. Nagy bitszélességű karaktertípus 270
F1.8. A mutable kulcsszó 271
F1.9. Konstruktorok szigorú paraméterezése 271

F2. Objektum-orientált megközelítésű függvények 273
F2.1. Komplex aritmetika 273
F2.2. BCD aritmetika 276

F3. Függvények szöveges üzemmódban 279

F4. Függvények grafikus üzemmódban **289**

F5. Borland C++ include fájlok **321**

F6. Általános könyvtári függvények **325**

F7. A lemezmelléklet használata **369**

Irodalomjegyzék **371**

Tárgymutató **373**

Köszönetnyilvánítás

Ezúton szeretnénk köszönetet mondani ***Tóth Bertalannak***, a *Gépészmérnöki Kar Informatikai Laboratórium* vezetőjének az Objektum-orientált fejezethez adott tanácsaiért.

Köszönetet mondunk ***Dr. Gyenes Károlynak*** és ***Dr. Komócsin Zoltánnak*** a *Közlekedésmérnöki Kar Számítástechnika* tárgy előadóinak a kézirat grafikus és animációs feladataiban nyújtott segítségéért.

Köszönetet szeretnénk mondani ***Szűcs Gábornak***, aki lelkiismeretesen lektorálta a kéziratot, ötleteivel és tanácsaival hozzájárult a könyv sokrétűségéhez.

Bevezetés

Az AT&T Bell Laboratóriumban dolgozó ***Bjarne Stroustrup*** nevéhez fűződik a C++ nyelv kidolgozása. A 70-es évek elején ugyanitt fejlesztették ki a C nyelvet. Így érhető, hogy a tíz évvel később kifejlesztett C++ nyelv a C nyelvre épült. A C nyelv ismerete mindenképpen szükséges a C++ nyelv megismeréséhez, mivel az felülről kompatibilis az eredeti C nyelvvel.

A C++ nyelv a C nyelv **struct** adatszerkezetére épült **class** osztályszerkezettel való bővítése lehetővé tette az objektum-orientált programozás (OOP) megvalósítását. A 90-es évek közepére az egész világon általánossá vált a problémák objektum-orientált megközelítése.

Egy objektum-orientált programozási nyelv sokkal strukturáltabb, modulárisabb és absztraktabb, mint egy hagyományos programozási nyelv. Míg a hagyományos programozási nyelvek használata során az adatok csak másodlagos szerepet töltenek be a rajtuk elvégzendő műveletekkel (függvényekkel) szemben, addig az objektum-orientált nyelvben az adatokat és adatokon elvégzendő műveleteket egyenrangúan, zárt egységben kezeljük. Ezeket az egységeket objektumoknak hívjuk. Az adatok (*adattagok*) és az adatokat kezelő függvények (*tagfüggvények, metódusok*) egységbezárása (*encapsulation*) nagyon fontos sajátossága az objektum-orientált nyelveknek. A C++-ban az objektumoknak megfelelő tárolási egység típusát osztálynak (***class***) nevezzük. Az objektum tehát valamely osztálytípussal definiált változó, amelyet más szóhasználattal az osztály példányának nevezünk (*instance*).

Az így kialakított osztályok egy további, a nagy méretű programok kialakításánál fontos lehetőséggel, az adatrejtés (*data hiding*) képességével rendelkeznek.

A C++ nyelvben egy osztálytípus a nyelv szerves részévé tehető, felhasználva az *operator overloading* (*operátor átdefiniálás*) mechanizmusát

Egy objektum-orientált program készítésénél a feladat megoldásához meg kell terveznünk, hogy az objektumok milyen adattagokkal rendelkezzenek, és hogy az adattagokon mely tagfüggvényekkel milyen műveleteket végezzünk. Az objektumok inicializálására konstruktor, illetve lebontására destruktor függvényeket használhatunk.

Az objektum-orientált nyelvek másik fontos sajátossága az öröklődés (*inheritance*) lehetősége. Az öröklődés azt jelenti, hogy a meglévő osztály(ok)ból kiindulva újabb osztályt építhetünk fel, amely örökli a felhasznált osztály(ok) adattagjait és tagfüggvényeit. A C++-ban azt az osztályt, amelyből az új osztályt származtatjuk ős, vagy alap (*base*) osztálynak, míg az új osztályt származtatott (*derived*) osztálynak nevezzük. A *multiple inharitance* azt jelenti, hogy az új osztály származtatása során több alaposztályból indulunk ki.

A származtatott osztály örökli az alaposztály(ok) tulajdonságait (adattagjait és tagfüggvényeit), ezek azonban meg is változtathatók, lehetőség van:
- új tagok hozzáadására,
- a tagfüggvények újradefiniálására (*redefine*),
- az örökölt tagok elérhetőségének megváltoztatására.

Egy statikus osztályszerkezet a zártság tulajdonság érvényesülése miatt nem minden esetben teszi lehetővé a tagfüggvények újradefiniálását a származtatás során. Ekkor a többrétegűség (*polymorphism*) mechanizmusát kell alkalmaznunk, amely például a virtuális (***virtual***) tagfüggvények formájában valósul meg a C++-ban. Egy osztályhierarchián belül a virtuális tagfüggvények teljesen újradefiniálják az alaposztály ugyanilyen nevű tagfüggvényét. Ez azt jelenti, hogy attól függően, hogy a programunk futása során az objektum-hierarchia mely szintjén vagyunk, mindig más-más művelet (tagfüggvény) kerül végrehajtásra. Ily 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 összerendeléstől (*early binding*). A virtuális tagfüggvények használata alapvető követelmény minden objektum-könyvtártól.

Az osztályok nemcsak, hogy támogatják az adatrejtést, hanem lehetővé teszik az adatok inicializálását, a felhasználó által definiált típusokra vonatkozó implicit típuskonverziót, dinamikus típusok használatát, a felhasználó által vezérelt memóriakezelést és biztosítják az operátorok átdefiniálásának lehetőségét is.

A C++ nyelv a Borland cég legújabb, 5.02 verziószámú implementációja során a korábbi C(++) megvalósításokhoz képest további lehetőségekkel is bővült:
- paraméterezett típusok, osztályok és függvények (*template*) használata,
- kivételek kezelése,
- *inline* helyettesítésű függvények,
- alapértelmezésű függvény argumentumok,
- függvény *overloading*,

– referencia típusok, stb.

Feltételezzük, hogy az Olvasó jártas a C nyelvű programozásban. Azoknak az Olvasóknak, akik még bizonytalanok a C nyelv alapjaival, ajánljuk a *Programozzunk C nyelven kezdőknek és középhaladóknak* című könyvünket.

A C++ nyelven való programírás felzárkózásához nyújt segítséget a C++ program lépésről-lépésre fejezet, amely példákon keresztül átveszi a szükséges elméletet a továbbhaladáshoz.

A C++ mint objektum-orientált nyelv fejezet foglalkozik valójában az OOP (Objektum-Orientált Programozás) alapelveivel és egy példasorozaton mutatja be a nyelv lehetőségeit.

A feladatokat a Windows 3.x alatt futó Borland C++ 3.1 és a Windows'95 alatt futó Borland C++ 5.02 rendszerében teszteltük a fordító DOS alatt működő változatával.

További fejezetek különféle grafikus feladatokon belül az animációt tűzték ki célul.

A függelék bemutatja a Windows 95 alatt futó C++ 5.02 fordító DOS verziójának specifikációját

A mintapéldák kialakításánál szem előtt tartottuk a hordozhatóságot, így programjaink nagy része változatlan formában vagy csak minimális módosítással gond nélkül lefordítható és futtatható nem PC típusú számítógépeken is.

A lemezmellékleten külön könyvtárban találhatók a Windows 3.x illetve a Windows'95 alatt futó programváltozatok.

Könyvünk szövegének és mintapéldáink elkészítése során nagyban támaszkodtunk a BME Villamosmérnöki és Informatika Karán gyűjtött sokéves oktatói tapasztalatunkra.

F-SECURE Anti-Virus

F-SECURE
ANTI-VIRUS

A vállalati szintű vírusvédelmi megoldás

- Teljeskörű hálózati adminisztráció
- CounterSign™ technológia - két víruskereső egyben!
- Új technológia a makróvírusok ellen
- DOS, Windows, Windows 95, Windows NT és Novell NetWare rendszerekhez
- SMTP levelezés ellenőrzése
- Firewall-1 integráció

Az F-PROT és az AVP víruskeresőivel!

2F Szervezési, Számítástechnikai és Szolgáltató Kft.

1016 Budapest, Hegyalja út 5. Tel: 212-7141, 212-7142 Fax: 212-7143
e-mail: info@2fkft.com http://www.2fkft.com/ BBS: 319-0466

1. C++ program lépésről-lépésre

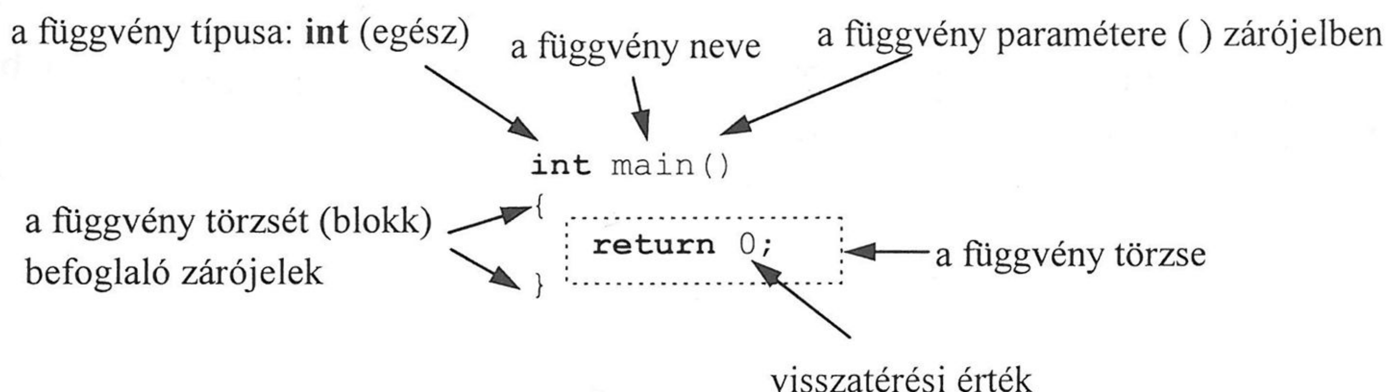
A C++ egy objektum-orientált nyelv, amelyet a C nyelvre alapoztak. Ebben a fejezetben röviden összefoglaljuk a legfontosabb alapismereteket egyszerű C++ programok bemutatásával, tárgyalva a C és a C++ nyelv különbségeit. Mint tudjuk, a Borland C++ rendszerben egy C programot .c kiterjesztésű, addig a C++ programokat .cpp kiterjesztésű forrásfájlban hozzuk létre. Egy C programot fordíthatunk C++ fordítóval, de C++ programot C fordítóval nem lehet lefordítani. Például a C nyelvben dupla deklaráció hibajelzést kapunk, ha egy függvény kétféle prototípussal szerepel a programban, ez a C++ nyelvben legális, mivel ez átdefiniált (*overloaded*) függvényt jelent.

1.1. Az első C++ program

Egy C++ program egy vagy több függvényből áll. A függvények közül - hasonlóan a C programokhoz - egyet kötelezően a ***main*** névvel kell ellátni. A program végrehajtása ennek a függvénynek az aktiválásával kezdődik.

A ***main*** függvénynél a kerek nyitó és záró zárójelet akkor is ki kell tenni, ha a függvénynek nincs paramétere, a törzsét pedig a nyitó és csukó kapcsos zárójelek fogják közre. A függvény jellemzője a visszatérési értéke, melynek típusát a függvény fejlécében, a függvény neve előtt kell megadni. A visszatérési értéket a **return** utasítás utáni kifejezés kiértékelésével határozza meg a függvény.

A legegyszerűbb C++ program egy üres ***main*** függvény (prg_0.cpp):



A **void** (üres) adattípust is használhatjuk a ***main*** függvénynél, ha nem használjuk a visszatérési értéket (prg_1.cpp):

```
void main()
{
}
```

A prg_2.cpp program szöveget ír a képernyőre:

```
#include <iostream.h>
void main()
{
  cout << "Első C++ program.";
}
```

A program futásának eredménye:

```
Első C++ program.
```

Hasonlóan a C programokhoz, a C++ nyelvben a szabványos I/O műveletek végzésére használt a *cin* és *cout* adatfolyam (*stream*) objektumok nem részei a C++ nyelv definíciójának. A képernyőre való kiíráshoz szükséges a *cout* adatfolyam (*stream*) objektum definiálás, amely az *iostream.h* állományban található.

1.2. Megjegyzések

A C++ megengedi az alábbi megjegyzéseket:

```
int /* lokális deklaráció */  sum  /*  összeg */;
```

melynek jelenléte a fordítás számára:

```
int sum;
```

A //-rel kezdődő C++ megjegyzés csak egy soros lehet. Kezdődhet a sor bármely karakterén és a sor végéig tart:

```
// ez már C++ megjegyzés

      int n; // az adatok száma
```

1.3. A C++ nyelv kulcsszavai

Az alábbiakban összefoglaljuk a kulcsszavakat, amelyekkel a C++ nyelv bővült. Az 1.1. táblázat a nyelv Borland C++ 5.0 implementációja szerinti kulcsszavakat tartalmazza (a vastaggal jelölt kulcsszó jelzi az előző változathoz képesti bővítést).

1.1. táblázat

asm	**mutable**	this
bool	**namespace**	**throw**
catch	new	**true**
class	operator	**try**
const_cast	private	**typeid**
delete	protected	**typename**
dynamic_cast	public	**using**
explicit	**reinterpret_cast**	virtual
false	**_rtti**	**wchar_t**
friend	**static_cast**	
inline	template	

1.4. A C++ nyelv azonosítói

A C++ nyelvű programban a változókra, függvényekre, címkékre stb. névvel hivatkozunk. A nevek (azonosítók, szimbólumok) megválasztása lényeges része a program írásának.

A programok változókat használnak az információ tárolására. Mielőtt használnánk egy változót, azt előzőleg deklarálni kell. A változó deklarálásához meg kell adnunk a változó nevét és típusát.

A változó nevében szerepelhetnek betűk (a..z, A..Z), számok (0..9) és _ (aláhúzás). Megjegyezzük azonban, hogy a változó nevének betűvel vagy aláhúzással kell kezdődnie. A hosszát a különböző fordítók korlátozzák, a Borland C++ az azonosítók első 32 karakterét veszi figyelembe, ez alapján tesz közöttük különbséget.

A C++ nyelv különbséget tesz a kis- és a nagybetűk között. Így különböző azonosítást jelent az *Adat, aDat* vagy *adaT*. Változó például: *X_adat, t2, tomb_valos*.

1.5. Alaptípusok

A beépített alaptípusok az alábbi kulcsszavakkal használhatók:

1.2. táblázat

Alaptípusok	Tárolása
char	karakter
int	egész szám
float	lebegőpontos szám
double	dupla pontosságú szám
Módosítók	
short	rövid
long	hosszú
signed	előjeles
unsigned	előjel nélküli

A **long** és a **short** használható az **int** típussal. A **long** és a **short** egymagában is típus értékű, **long int**-et illetve **short int**-et jelent.

A **signed** vagy **unsigned** módosító a **char**, **short, int** és a **long** típussal használható. Ha a **signed** vagy **unsigned** módosítókat önmagukban használjuk, akkor a jelentésük **signed int** és **unsigned int** lesz.

A **float** és a **double** 32 és 64 biten tárolja a lebegőpontos számot, a **long double** pedig 80 biten. Megjegyzendő, hogy a **long double** típus a Borland C++ implementáció sajátja, más fordítók nem feltétlenül ismerik.

1.6. Konstansok

A C++ nyelv megkülönbözteti a numerikus és a szöveges konstans értékeket. A numerikus konstansok alatt mindig valamiféle számot értünk, míg a szöveges konstansokat sztring literálnak hívjuk. A Borland C++ nyelvben egész, lebegőpontos, karakteres és felsorolt konstansokat használhatunk.

1.6.1. Egész konstansok

Az egész konstansok lehetnek decimális (10-es), oktális (8-as) vagy hexadecimális (16-os) számrendszerbeli számok.

A *decimális konstansok* megengedett értékhatára (abszolút értékben) 0 és 4,294,967,295 között van, ezen határt meghaladt érték csonkításra kerül.

```
int j = 10;          /* decimális 10 */
int k = 010;         /* decimális 8 */
int n = 0;           /* decimális 0 vagy oktális 0 is lehet */
```

Az *oktális konstansok,* melyek 0-val kezdődnek, megengedett értékhatára 0 és 037777777777 között van, ezen kívül az érték csonkításra kerül.

A 0x-szel (vagy 0X) kezdődő *hexadecimális konstansok* megengedett értékhatára 0x és 0xFFFFFFFF között van, ezen határt meghaladt érték csonkításra kerül.

Az előjel nélküli egész szám használatánál az **u** (**unsigned**) vagy **U** módosítót kell kitennünk:

125U, 072u, 0xFFFFu.

Hosszú egész használatánál a **l** (**long**) (kis **L**) vagy **L** módosító szükséges:

12567L, 3487653l

Előjel nélküli hosszú egész használatánál **LU** vagy **UL** (**unsigned long**) módosítókat kell kitenni:

23467UL

Borland C++-ban használható egész konstansok értékhatárai:

Decimális konstansok		
0	32,767	**int**
32,768	2,147,483,647	**long**
2,147,483,648	4,294,967,295	**unsigned long**
	> 4,294,967,295	*csonkításra kerül*
Oktális konstansok		
00	077777	**int**
010000	0177777	**unsigned int**
02000000	017777777777	**long**
020000000000	037777777777	**unsigned long**
	>037777777777	*csonkításra kerül*

Hexadecimális konstansok

0x0000	0x7FFF	**int**
0x8000	0xFFFF	**unsigned int**
0x10000	0x7FFFFFFF	**long**
0x80000000	0xFFFFFFFF	**unsigned long**
	> 0xFFFFFFFF	*csonkításra kerül.*

1.6.2. Lebegőpontos konstansok

A Borland C++-ban ábrázolható valós számok bitmérete és abszolút értékének ábrázolási határai:

1.3. táblázat

Típus	Méret(bitekben)	Alsó határ	Felső határ
float	32	3.4×10^{-38}	1.7×10^{38}
double	64	1.7×10^{-308}	3.4×10^{308}
long double	80	3.4×10^{-4932}	1.1×10^{4932}

A lebegőpontos konstansok mérete és határai azonosak a 16 és 32 bites adattípusok esetén is.

1.6.3. A 32 bites adatok típusai, méretei és határai

1.4. táblázat

Típus	Méret(bitekben)	Alsó határ	Felső határ
unsigned char	8	0	255
char	8	-128	127
short int	16	-32,768	32,767
unsigned int	32	0	4,294,967,295
int	32	-2,147,483,648	2,147,483,647
unsigned long	32	0	4,294,967,295
enum	32	-2,147,483,648	2,147,483,647
long	32	-2,147,483,648	2,147,483,647

A 16 bites fordítónál az **enum**, **int**, **unsigned int** mérete 16 bit.

1.6.4. A karakter konstansok

A karakter konstansok egyszeres idézőjelek (' =aposztróf) közé zárt egy vagy több karaktert tartalmazó karaktersorozatok: `'w'`, `'3'` ,`'#'`, `'01'`

Az egyetlen karaktert tartalmazó karakter konstansok által képviselt számérték a karakter kódja. Több karaktert tartalmazó konstansok számértéke implementáció függő.

A C nyelvben az egyetlen karakter konstans típusa **int**, míg a C++ nyelvben **char**. A Többkarakter konstans mind a két nyelvben (C és a C++) **int** típusú.

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

Aa 1.5. táblázat bemutatja a használható *escape* szekvenciákat.

1.5. táblázat

Szekvencia	Értéke	Karakter	Művelet
\a	0x07	BEL	csengő
\b	0x08	BS	backspase
\f	0x0C	FF	lapdobás (formfeed)
\n	0x0A	LF	új sor
\r	0x0D	CR	kocsi vissza (carrige return)
\t	0x09	HT	tabulátor (vízszintes)
\v	0x0B	VT	függőleges tabulálás
\\	0x5c	\	backslash
\'	0x27	'	aposztróf
\"	0x22	"	dupla aposztróf
\?	0x3F	?	kérdőjel
\O		bármi	karakter oktális kóddal megadva
\xH		bármi	ASCII karakter hexadecimális kóddal megadva
\XH		bármi	ASCII karakter hexadecimális kóddal megadva

Az operációs rendszer útvonalához szükséges az ASCII backslash (\), melyet a C++ nyelvben a \\ használatával érhetünk el. Gyakran használjuk a 0 kódú karaktert (sztring vége jel), amit egyszerűen '\0' alakban adhatunk meg.

1.6.5. Sztring konstansok

A sztring konstans kettős idézőjelek közé zárt karaktersorozatot jelent.

```
"Sztring konstans megadása."
```

A hosszú sztringeket összefűzhetjük a sor végén " lezárás alkalmazásával. A sztring konstans tartalmazhat *escape* szekvenciákat, például az új sort (*\n*):

```
"Ez az szöveg"
" tovább "
"folytatódik\n, de új sorban "
"is írható."
```

Használhatunk \ (*blackslash*) folytató karaktert is a sor végén a sztringek összefűzésére:

```
"Ez az első sor.\
Ez a második sor."
```

1.6.6. Enum konstansok

Egész értékeket használ az ún. felsorolt (**enum**) típus is. Az **enum** deklarációban az azonosítóknak egyedinek kell lenni. Ha nem adunk kezdőértéket, akkor a sorszám nullától egyesével nő. Negatív értéket is lehet kezdőértéknek adni.

```
enum  nyelvek {angol, olasz, spanyol, magyar};
```

Ekkor az angol értéke 0, olasz 1, spanyol 2 és a magyar 3 lesz. Példa nem 0 kezdőértékek megadására:

```
enum nyelvek {angol = -1, olasz = 2; spanyol = 1, magyar = 5};
```

1.7. Tárolási osztályok, hatáskörök

A legtöbb C++ program több, külön fordítható modulból épülhetnek fel. Jogosan merül fel a kérdés, hogy az egyik modulban definiált tárolási egységre tudunk-e hivatkozni egy másik modulból, másrészt kíváncsiak lehetünk arra is. hogy milyen a program egyes részeinek hierarchiája, "elrejthetjük-e" változókat más modulok elől, stb. Ahhoz, hogy ezekre a kérdésekre válaszolni tudjunk, tovább kell boncolgatnunk a C++ programok szerkezetét. A modulokban van-

nak a kódgeneráló egységek, a függvények. Minden függvény törzse egy blokkból áll, amelyen belül újabb blokkokat alakíthatunk ki. (A blokk más elnevezése: összetett utasítás). A blokkszerkezet tehát a modulszerkezettel ellentétben hierarchikus, hiszen minden blokknak lehet albokkja, és minden alblokk alá van rendelve az őt magában foglaló blokknak; míg a modulok azonos hierarchia szinten vannak, tehát egymás mellé vannak rendelve. Minden blokk két részre osztható: deklarációkra és adatdefiníciókra, valamint végrehajtható utasításokra (ez utóbbiak tartalmazhatják az alblokkokat). Mivel blokk belsejében nem lehet függvénydefiníció, ezért a C++ nyelvben minden kódgeneráló tárolási egység azonos hierachia-szintű (mint a modulok). Természetesen a függvények között van egy logikai függőségi rendszer, hiszen a függvények egymást adott szisztéma szerint hívják (például, ha egy ***fv1*** hív egy ***fv2***-t, akkor ritkán fordul elő, hogy ***fv2*** is hívja ***fv1***-et). Logikailag mindig van egy vezető függvény, amelyik a program indulásakor legelsőként kapja a vezérlést. A C illetve C++ nyelvben ennek a vezető függvénynek a neve kötelezően a ***main*** (lásd 1.1. alfejezetet).

1.7.1. Függvények tárolási osztályai

A kódgeneráló tárolási egységek elérhetőség szerint kétfélék lehetnek: vannak csak a definíciót tartalmazó modulban elérhető és a mindenhonnan hívható függvények. Az előbbieket **static** tárolási osztályúaknak (**static** *storage class*), az utóbbiakat **extern** tárolási osztályúaknak nevezzük. Ha egy adott funkciót megvalósító függvény több, mások számára érdektelen segédfüggvényt használ, akkor ezeket összegyűjthetjük egy modulba, és egy kivételével – aminek hívása kezdeményezi a kérdéses feladat végrehajtását – a többit **static** tárolási osztályúnak választjuk. Ilyen módon lehetővé tesszük, hogy egy – gyakran más programozó által írt – másik modulban ugyanolyan azonosítókat konfliktus nélkül lehessen használni.

1.7.2. Változók élettartama és hatásköre

A tárterületet foglaló tárolási egységeket élettartam és hatáskör szerint csoportosíthatjuk. A C++ nyelvben az élettartam kétféle lehet: a program egész futására kiterjedő, illetve egy bizonyos blokkban tartózkodás idejére korlátozódó. Az előbbi csoportba tartozó változókat statikusoknak, az utóbbiba tartozókat dinamikusoknak vagy automatikusoknak (**auto**) nevezzük. A statikus változók

tehát már a program indulásának a pillanatában léteznek, ily módon – mint látni fogjuk – kezdőértékkel is rendelkeznek, míg a dinamikusoknak a létrehozása és megszüntetése futás közben történik, az egyes blokkokba való belépés, illetve kilépés alkalmával. Ugyanabba a blokkba való többszöri belépéskor a dinamikus változóknak mindig egy új, friss készlete áll rendelkezésre, amelyekben nyoma sincs azoknak az értékeknek, amelyeket az utolsó kilépéskor bennük hagytunk. Éppen ezért a dinamikus változókat használó függvények rekurzív módon is hívhatók. A dinamikus tárkezelés másik előnye, hogy sokkal jobban gazdálkodik a memóriával, hiszen egy blokkból kilépve, a felszabadított tárterület felhasználhatóvá válik a következő blokk számára, azaz a változók egy része a tárban – időeltolással – átfedheti egymást.

Változók élettartama, láthatósága és tárolási osztálya

1.6. táblázat

Élettartam	Láthatóság	Deklarátor helye	Tárolási osztály
statikus	globális	bármely modulban minden blokkon kívül	**extern**
statikus	modulra lokális	adott modulban minden blokkon kívül	**static**
statikus	blokkra lokális	adott blokkban	**static**
dinamikus	blokkra lokális	adott blokkban	**auto, register**

Hatásköri csoportosítás szerint léteznek teljesen globális (mindenhonnan elérhető), modulra nézve lokális (csak az adott modulból használható) és blokkra nézve lokális (csak az adott blokkon belül elérhető) változók. A C++ nyelvben az első két csoportba csak statikus helyfoglalású változók tartozhatnak, az utolsóra nincs ilyen megkötés. A blokkban a lokális változók hatásköre kiterjed az egész blokkra, beleértve az alblokkokat is, azonban ha egy ugyanilyen azonosítójú változót definiálunk egy alblokkban, akkor az alblokkban való tartózkodás idejére az új definíció felülbírálja az előzőt, és új változót hoz létre, úgymond "elfedi" az eggyel magasabb hierarchiaszinten definiált változót. Az alblokkból való kilépés után a magasabb blokkban való definíció fog újra érvényesülni. Ugyanez a helyzet akkor is, ha egy blokkban egy modulra lokális, vagy teljesen globális változót definiálunk újra.

Egy változó deklarációja illetve definíciója a változó típusán kívül megadja azt is, hogy a fenti csoportok közül hová kerüljön. Ezt két dolog szabályozza: hol

szerepel az adott deklarátor, és hogy milyen tárolási osztályt írunk elő. Ezeket az 1.6. táblázatban foglaltuk össze.

Mielőtt a deklarátorok konkrét formájára rátérnénk, szeretnénk megvilágítani a **register** tárolási osztályt. Ez lényegében az **auto**-val egyezik meg, az egyetlen eltérés, hogy ennek használatával ráirányítjuk a fordítóprogram figyelmét az adott változóra: várhatóan sokszor fogunk futás közben hozzáfordulni, ezért optimális tárolás megválasztását kérjük (lehetőleg regiszterbe). Ez nem kötelezi a fordítóprogramot semmire, és az, hogy mennyiben tudja teljesíteni igényüket, attól is függ, milyen típusú változóról van szó. Csak sorszámozott, vagy pointer típusú változó lehet **register** tárolási osztály. Mivel az **int** bitszélességét mindenhol a gépi szó méretére választják, ezért ennél rövidebb méretű egész változónál a **register** előírás nem célszerű. A **register** tárolási osztály előírása az egyik olyan eset, amikor az **int** használata ajánlott. Az egész típusú változókon kívül **register** tárolási osztályú változónak ajánlhatók még a pointerek.

1.7.3. Egyszerű deklarátorok

A adatdefiníciók legegyszerűbb formája a következő:

tárolási osztály spec. típus spec. azonosítólista;

Ha a tárolási osztályt nem specifikáljuk, akkor a következő két eset lehetséges: ha minden blokkon kívül vagyunk, akkor a változó globális lesz, míg ha valamelyik blokk belsejében vagyunk, akkor a legbelső befoglaló blokkra nézve lokális, dinamikus változó jön létre (tehát **auto**, ezért ezt a kulcsszót nem szokás sehol kitenni). Amennyiben a definícióból a típusmegadás hiányzik, akkor a fordító **int**-nek tételezi fel, ez akkor is igaz, ha típusmódosító kulcsszó (**short**, **long unsigend**) mellett nem áll alaptípus.
Az azonosítólista elemeit vesszővel választjuk el, az utasítást pontosvessző zárja:

```
int i,j;
unsigned d;
static unsigned short db;
double x,y;
long double w;
```

Megjegyzendő, hogy egy változó definíció deklarációs értékű. (A definíció létrehozza a szóban forgó változókat, a deklaráció csak a főbb jellemzőit – azonosító, név – adja meg. E különbségről később, az **extern** kulcsszó kapcsán még szót ejtünk.)

Statikus helyfoglalású változó definíciójában előírhatunk kezdőértéket is úgy, hogy a definiált azonosító után egyenlőségjellel elválasztva kerül az inicializáló kifejezés. Kezdőértéket adhatunk szimbólumdefinícióban (lásd 1.18.1. alfejezet) megadott szimbólummal is.

```
#define  EOS '\0'   // karaktersorozatot lezáró \0
```

Például:

```
static char c = EOS;
```

Ennek hatására az adott változót a fordító eleve úgy helyezi el, hogy benne van a megfelelő kezdőérték, tehát ez az értékadás futási időben nem igényel időt. Ehhez azonban természetesen az szükséges, hogy az inicializáló kifejezés ún. állandó kifejezés legyen, azaz fordítási időben kiértékelhető (nem függhet más változók értékétől). A C(++) nyelv garantálja, hogyha egy statikus helyfoglalású változónak nem adunk kezdőértéket, akkor az csupa 0 bittel lesz inicializálva (azaz a fenti példában felesleges a kezdőérték megadása.)

Dinamikus változók definíciója után is lehet kezdőértéket írni, azonban ez csupán rövidítés, mert a helyfoglalás után a "kezdőérték" ugyanolyan módon kerül a változóba, mintha értékadó utasítást használnánk. Tehát futás közben kap értéket és emiatt az itt használt kifejezésre nincs semmilyen megkötés, még függvényhívásokat is tartalmazhat.

```
int i = 0,j = fv(1);
```

Az előbbiektől kissé eltérő formájú a felsorolt (**enum**) típusú változók deklarálása. Az általános alak:

tárolási osztály spec. ***enum*** *típuscímke {elemlista} azonosítólista;*

A – nem kötelezően megadott – típuscímke az adott felsorolt típus megnevezése lesz, ezt követően már használható az alábbi forma.

tárolási osztály spec. ***enum*** *típuscímke azonosítólista;*

Az elemlista az adott típusba tartozó elemeket adja meg, az azonosítólista a definiált változókat nevezi meg, ha elmarad, akkor csak a típust deklaráljuk későbbi definíciók kedvéért. Példa a felsorolt típusú változók megadására:

```
static enum nap {
                  hetfo, kedd, szerda, csutortok,
                  pentek, szombat, vasarnap
               } ma;
enum nap tegnap, holnap;
```

Az Olvasó talán észrevette, hogy az **extern** kulcsszót eddig nem használtuk. Nos, ez azért van, mert a globális változók definíciójánál nem is szabad kiírni. Ezzel a kulcsszóval azt jelezzük, hogy nem definiáljuk az adott változót, hanem deklaráljuk. A globális változókat más modulokban is használhatjuk, de ahhoz, hogy a fordítóprogram az azonosítót megfelelően értelmezhesse, azt számára deklarálni kell. Az **extern** kulcsszó szerepe tehát jelezni azt, hogy az azt követő deklaráció valami olyan tárolási egységre vonatkozik, amelyet egy másik – az aktuális modulra nézve külső modulban definiáltunk.

Általános szabály, hogy csak olyan változót szabad a C++-ban használni, amelyik az adott modulban – akár fejléc (*include*) fájlokon keresztül – előzőleg deklarálva van. Egy modulra nézve külső globális változók deklarációja formailag megegyezik definíciójukkal, a tárolási osztályuk **extern**, és természetesen nem szerepelhet kezdőérték adás. Például:

```
extern short unsigned kulso_valtozo;
```

Egydimenziós tömbváltozó (*array*) megadása a név után [ ]-be írt elemszámmal (felső indexhatár+1) történik, például a

```
float vektor[20];
```

egy 20 elemű lebegőpontos elemből álló *vektor* nevű tömböt definiál. Hivatkozni az egyes tömbelemekre tetszőleges kifejezéssel lehet. A tömbindexek a C(++) nyelvben mindig 0-tól indulnak és a megadott méretnél eggyel kisebb értékig mehetnek, tehát példánknál *vektor[0]*-tól *vektor[19]*-ig. Statikus tömbök inicializálhatók is, { } zárójelek közötti elemlistával, például:

```
static short paratlan [] = {
                             1, 3, 5, 7, 9,
                            11, 13, 15, 17, 19
                           };
```

Figyeljük meg, hogy ez esetben az indexméret el is maradhat, mert a fordítóprogram a kezdőérték száma alapján foglalja le a szükséges tárterületet. Karaktertömbök sztringek segítségével is inicializálhatók:

```
static char hiba[] = "Hibás név!";
```

Itt a hiba tömb 11 elemű lesz, *hiba[10]* értéke EOS lesz. Ha egy sztring nem kezdőértékként jelenik meg a programban, akkor egy statikus helyfoglalású, az adott karakterekkel (+ az EOS) inicializált tárterületet foglaló egység lesz, azonosító nélkül.

Mutatók definiálását (vagy deklarálását, ez ebből a szempontból közömbös) a mutatott típus definíciójából vezetjük le. Ha az

```
unsigned short abc;
```

definiálja az abc-t, mint egy rövid, előjel nélküli egész változót, akkor az

```
unsigned short *p;
```

definiája a *p*-t, mint egy rövid előjelnélküli egészre mutató pointert. A *p* mutató dinamikus helyfoglalással kerül tárolásra.
A tömbökre és a mutatókra később részletesebben, példákkal visszatérünk az 1.13. alpontban.

1.7.4. Módosító jelzők

Kernighan és Ritchie "A C programozási nyelv" könyvében 3 módosító jelzőt definiál, ezek a **short**, **long** és az **unsigned**. A Borland C++ az ANSI szabványajánlásnak megfelelően további hármat valósít meg, ezek a **signed**, **const** és **voletile**.

A signed módosító jelző

A **signed** jelző azt állítja egy adott objektumról, hogy nem **unsigned**, tehát nem előjeles értelmezésű. Ennek szerepe elsősorban dokumentatív, illetve szimmetriát biztosít. Ha azonban az alapértelmezésbeli karaktertípust előjel nélkülire választjuk, akkor csak ennek segítségével definiálhatunk előjeles karaktereket. A **signed** önmagában **signed int** típust határoz meg, ugyanúgy, mint ahogy az **unsigned** önmagában **unsigned int** típust jelent.

A const módosító jelző

Ha egy inicializált változódefiníció elé kitesszük a **const** módosító szót, akkor az így deklarált változóra úgy tekint a fordító, hogy annak értékét a későbbiek során már nem lehet megváltoztatni:

```
const double pi = 3.141592;
const int db = 100;
const int x[] = {2,5,10,4};
```

Az első definíció egy *pi* nevű duplapontosságú konstanst hoz létre, melynek értéke 3.141592, a második definíció egy *db* nevű egész konstanst hoz létre, értéke 100, a harmadik definíció pedig egy *x* azonosítójú, 4 elemű egész konstansokat tartalmazó tömböt eredményez. Mivel egy **const**-ként definiált változónak a programfutás során érték nem adható, mindig inicializáltan kell deklarálni! A **const** mindenütt állhat, ahol változót deklarálhatunk, így **auto** változók előtt elhelyezve lokális konstansokat kaphatunk. A **const** kulcsszó tulajdonképpen egy típusmódosító, amely azt mondja, hogy a hatásköre alatt létrehozott adott típusú kifejezés értéke nem változtatható meg. Ennek alapján értelmes az alábbi deklaráció is.

```
static char *private[ ] =
{
    "Ebben a privat"
    "tombben sok-sok "
    "uzenet van. "
}
const char * get_message(int i)
{

  return private[i];
}
```

A fenti függvénydeklaráció arra szolgál, hogy mások számára lehetővé tegyük egy sztring olvasását, de a visszatérési értékként kapott pointeren keresztül az adott sztringet soha ne lehessen módosítani.

A voletile módosító jelző

A **voletile** jelző azt jelenti, hogy a tárolási egység elvileg bármelyik pillanatban módosulhat egy, az adott programtól független folyamat (például megszakítási rutin, stb.) által. A fordítóprogram a **voletile** jelző hatására az adott változót nem fogja a regiszterben tárolni, és az optimalizálás során sem fogja a többszörös vagy látszólag felesleges értékadásokat kiiktatni.

A C++ kiterjeszti a **voletile** módosító jelzőt az osztályokra és tagfüggvényekre. Ha **voletile** objektumot deklarálunk, akkor csak **voletile** tagfüggvényeket használhatunk.

A Borland C++ esetében a **volatile** jelző majdnem az ellentéte a **const**-nak.

A **volatile** módosító jelző használatára tekintsük a következő példát:

```
volatile int s;
void interrupt timer(void)
{
     s++;
}
void wait ( int interval)
{
    s = 0;
    while (s < interval)
      ;
}
```

A *timer* függvény megszakítás rutinként aktivizálódik valamely periodikus hardveresemény hatására. A *wait* függvény a megszakítási rutin által módosított *s* értékét ciklikusan lekérdezi. Nagyfokú optimalizálás esetén a fordítóprogram generálhat olyan kódot, amely a ciklusban az *s* értékét csak egyszer veszi figyelembe (mert az a ciklusban explicit módon nem változik meg), ezáltal végtelen ciklust eredményez. Ennek elkerülésére szolgál az *s* definíciójában a **volatile** jelző: a ciklus minden egyes futása alkalmával ki kell értékelni az *s* tartalmát, hiszen azt egy megszakítási rutin módosítja.

1.7.5. Típusdefiniáló (typedef) azonosítók

A **typedef** segítségével saját típust definiálhatunk, amely különösen hasznos összetett típusok létrehozása során. Ha egy azonosító definíciója mellé a **typedef**-et adjuk meg, akkor az azt jelenti, hogy valójában nem hozunk létre új dolgot, hanem az adott azonosítót - mint egy szimbólumot - megadott típus megnevezésére kívánjuk használni.
Például:

```
typedef  int    egesz;
typedef  double tomb[10];
typedef  float  *fptr;
```

Az *egesz* szimbólum **int** típust jelent. A *tomb* típus **double** pontosságú tömböt definiál, melyet a **double** elemi típusból hozunk létre egy ún. típusmódosító operátor segítségével. Ebben az esetben a [ ] tömbtípust képző operátort használtuk fel. Az utolsó példa egy mutatótípust - a **float** típusra mutató pointert - lát el azonosítóval. Ezt az új típust a * mutatótípust képző operátorral állítottuk elő a **float** elemi típusból.
A deklaráció a fenti szimbólumok felhasználásával:

```
egesz  i1, i2;
tomb   a, b;
fptr   p1, p2;
```

Típusmódosító operátorok:

1.7. táblázat

Operátor	Megnevezés	Jelleg
()	függvénytípust képző operátor	postfix
[]	tömbtípust képző operátor	postfix
*	mutatótípust képző operátor	prefix

Az új típust mindig valamilyen már meglévő típusból (elemi típusból, struktúrából, vagy **typedef**-fel már korábban definiált típusból) hozhatunk létre úgy, hogy megnevezzük az új típust, és erre az új típusazonosítóra alkalmazzuk az egyes típusmódosító operátorokat ((),[],*). A típusmódosító operátorok közül a * prefix operátor (a módosítandó típus előtt áll), míg a () és a [] operátorok postfix operátorok (a módosítandó típus után állnak).

1.8. A szabványos Input/Output használata

Értékeljünk ki egy olyan programot, amely a kör területét számítja ki, a kör sugarát a billentyűzetről olvassa be és az eredményt a képernyőre írja ki. Az io1.cpp program még a C programozási nyelvnél megismert *stdio.h* fejléc fájlt használja, a ***scanf*** függvénnyel olvas, és a ***printf*** függvénnyel ír a képernyőre.

```
/* io1.cpp */
#include <stdio.h>
void main()
{
 float r; const double pi = 3.141592;
   printf(" A kör sugara: ");
   scanf("%d",&r);
   printf(" A kör területe: %lf\n",r*r*pi);
}
```

A program futásának eredménye:

```
A kör sugara: 2
A kör területe: 12.566368
```

Ez a program valójában C és nem igazi C++ program, csak a program fájlának .cpp kiterjesztése miatt kell C++ fordítóval fordítani. A programban a képernyőre való íráshoz a ***printf*** adatfolyam kezelő függvényt, a billentyűzetről való olvasáshoz a ***scanf*** függvényt használtunk, melynek deklarációját (prototípusát) az stdio.h állomány tartalmazza. A programba az ***#include*** előfordító utasítással építettük be az stdio.h (**St**andar**d** **I**nput **O**utput) állományt. Ezen függvények nem tekinthetők a C nyelv részének, mivel könyvtári függvények.

A ***printf*** függvény a programozó által megadott formátum szerint ír ki. A formátumot az ún. formátumsztring határozza meg. A ***printf*** függvény általános formája:

```
int printf("formátum sztring",argumentumlista)
```

A formátumsztring a képernyőre minden átalakítás nélkül kiírásra kerülő karaktersorozatokat és az argumentumlistára vonatkozó konverziós előírásokat tartalmaz. A konverziós előírások részeit, az egyes részek szerepét a 1.8. táblázat mutatja.

1.8. táblázat

jel	*jelző*	*mezőszélesség*	*pontosság*	*méretmódosító*	*konverziós betű*
%	- # 0	15	.5	l	d

Jelző	
-	balra igazítás
+	a számoknál az előjel (+,-) mindig megjelenik
szóköz	előjel helyén a - vagy szóköz jelenik meg
0	a szóköz 0-val töltődik fel az adat előtt
#	a hexadecimális számok előtt 0x, az oktális számok előtt 0 előtag jelenik meg.

Egy konverziós előírásnak legalább a kezdő % jelet és a konverzió típusára utaló betűt (kódot) tartalmaznia kell, a többi rész el is maradhat.
Az alap konverziós előírásokat a 1.9. táblázat tartalmazza.

1.9. táblázat

Kód	argumentum típusa	formátum
`%c`	**int**	egyetlen karakter
`%d`	**int**	előjeles decimális egész
`%ld`	**long int**	előjeles hosszú decimális egész
`%f`	**float**	[-]ddd.dddd, ahol a tizedesjegyek száma az előírt pontosságtól függ. Fixpontos kiírási mód, alapértelmezés 6 tizedes, 0 pontosság esetén a tizedespont nem íródik ki.
`%lf`	**double**	hasonló a %f formátumhoz
`%e`, `%E`	**double**	[-]d.dddde±ddd, vagy [-]d.ddddE±ddd a tizedesjegyek száma az előírt pontosságtól függ
`%g`, `%G`	**double**	%e, %E, ha az exponens kisebb, mint -4, vagy a >= mint a pontosság, különben %f formátum. Nincsenek vezető nullák és szóközök.
`%o`	**int**	előjel nélküli oktális egész 0 előtag nélkül
`%s`	**char***	sztring karakterei '\0' (EOS) karakterig
`%x`, `%X`	**int**	hexadecimális egész 0x előtag nélkül
`%p`	**void***	mutató értéke

A ***scanf*** függvény első argumentuma a formátumsztring, amely a változókra vonatkozó konverziót tartalmazza, a további argumentum vagy argumentumok a változók, amelyek a beolvasott értéket kapják. Ahhoz, hogy a változó felvegye az értéket, a címét kell megadnunk, ezt az & (címe) operátor segítségével tehetjük meg.

A visszatérési értéke a sikeresen beolvasott adatok száma, ez lehetőséget nyújt az ellenőrzésre.

```
int scanf("formátumsztring", argumentumlista)
```

1.10. táblázat

konverziós karakter	Input adat	argumentum típusa
d	decimális egész	**int***
ld		**long** *
i	egész szám, oktális és hexadecimális is lehet	**int** *
o	oktális szám a 0 előtag nélkül	**int***
u	előjel nélküli decimális szám	**unsigned***
x	hexadecimális egész, 0x vagy 0X előtaggal	**int***
c	karakter beolvasása. A tagoló karakterek is beolvasásra kerülnek.	**char***
S	sztring olvasása tagoló karakterig	**char***
f,e,g	lebegőpontos számok beolvasása	**float***
E,G	Az L előtag esetén az argumentum típusa: %lf esetén az argumentum típusa	**double*** **long double***
p	a ***printf*** által kiírt formátumú mutató értékét olvassa be.	**void** **
n	a formátumig beolvasott karakterek számát adja vissza a függvény az argumentummal kijelölt egész (**int**) típusú változóban.	**int** *

A ***scanf*** formátumsztring elemei:
- a tagoló (*whitespace*) karakterek, ezeket a függvény figyelmen kívül hagyja,
- % karakterrel kezdődik a konverziós előírás, mely opcionálisan tartalmazhat:
  - * csillag karaktert, hatására a beolvasott szám eldobódik,
  - n mezőszélességet megadó decimális egész számot,
  - h,l,L opcionális méretkijelölő karaktert: h (**short**), l (**long** vagy **double**), L (**long double**).

Az io2.cpp program már igazi C++ program, mivel az *iostream.h* fejléc fájl lehetővé teszi a *cin* és a *cout* adatfolyam (stream) használatát, amellyel formátum megadása nélkül használhatjuk a standard I/O műveleteket.

```
/* io2.cpp */
#include <iostream.h>
void main(void)
{
 double r; const double pi = 3.141592;
  cout << " A kör sugara: ";
  cin >> r;
  cout << " A kör területe: " << r*r*pi << endl;
}
```

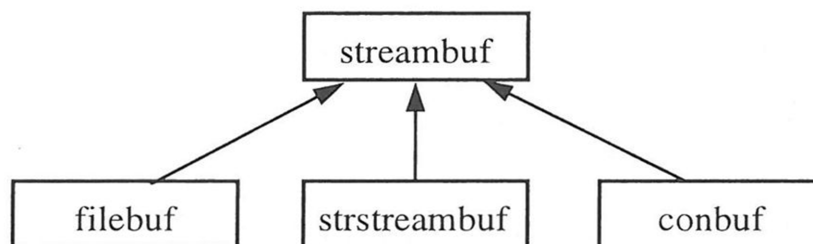
A program futásának eredménye:

```
A kör sugara: 2
A kör területe: 12.566368
```

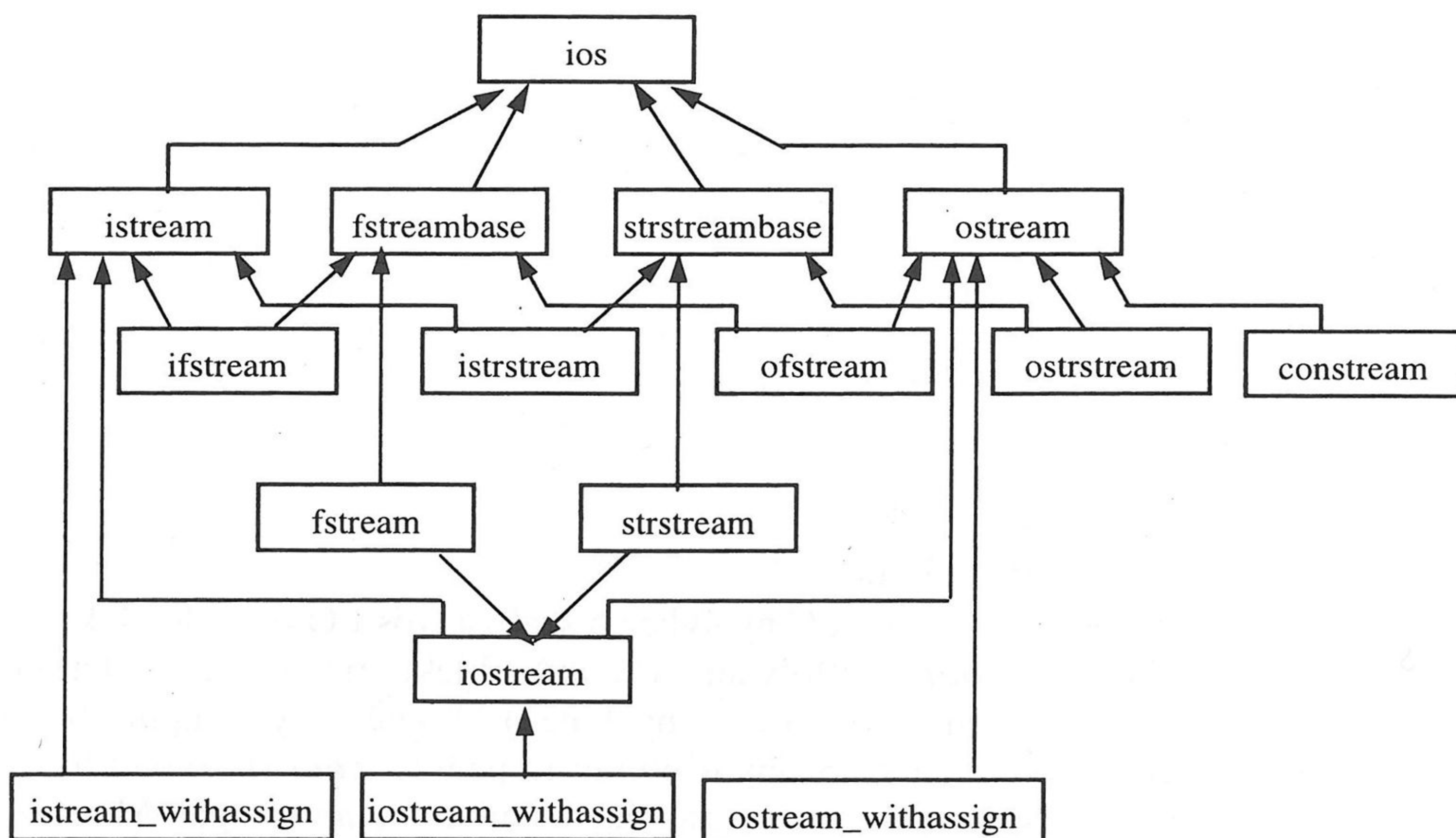
A következőket figyelhetjük meg:

- A C++ nyelvben hasonlóan a C nyelvhez a szabványos I/O műveletek kezelésére a ***cin*** és a ***cout*** adatfolyam (*stream*) objektumokat használhatjuk (amelyek azonban nem részei a C++ nyelvnek). A szabványos input elvégzésére a ***cin*** *stream*-et, a szabványos output-ra pedig a ***cout*** *stream*-et használjuk. Hibajelzésre a ***cerr***, az ún. szabványos hiba *stream* szolgál. Mindhárom *stream* definícióját az *iostream.h* deklarációs fájl tartalmazza.
- A C programhoz képest az I/O műveletek használata leegyszerűsödött, nem kell figyelni a megfelelő formátumok megadására, mivel a paraméterezést a fordítóprogram végzi el.
- A /* */ megjegyzések mellett a C++ nyelv a // jel utáni szöveget a sor végéig megjegyzésnek veszi, felhasználásával jól dokumentálható programot írhatunk. Megjegyezzük, hogy C programban a // jel használata kötelezi a programozót, hogy C++ fordítóval dolgozzon, különben hibajelzést kap.

A C++ nyelvben az I/O adatfolyam (összefoglalóan az iostreams) az ANSI C szabványos I/O könyvtárában lévő lehetőségeken túlmenően többet tartalmaz. A C++ adatfolyamoknak a használatával könnyebben olvashatunk és írhatunk adatokat a szabványos bemenetről ill. a szabványos kimenetre. A C++ adatfolyam deklarációja az *iostream.h* állományban található.



1.1. ábra Az **streambuf** osztály és a származtatott osztályai



1.2. ábra Az **io** osztály és a származtatott osztályai

Az *io* osztály egy pointert tartalmaz a *streambuf*-ra, melynek használatával hibaellenőrzéssel végrehajt egy formátumos I/O műveletet.

1.8.1. Szabványos output

A szabványos outputnál << operátor *ostream* osztályra kiterjesztett változatát használjuk az output műveletekre.

Írassuk ki az Eredmény szöveg mellett az *ered* változó tartalmát:

```
cout << "Eredmény: " << ered;
```

A *cout* a C-ben szokásos stdout megfelelője.
Az alábbi alaptípusok kiíratására használható a << operátor:

```
char, short, int, long, char*,
float, double, long double és a void*
```

A változók különféle formátumban történő kiírását egy speciális függvényhíváshoz hasonló operátorral az ún. *manipulátorral* tehetjük meg. A paraméteres manipulátorokat minden *stream* műveletnél meg kell hívni.

Az 1.11. táblázat tartalmazza a manipulátorral történő beállításokat és paramétereket.

1.11. táblázat

manipulátor	**művelet**
dec	Beállítja a decimális konverzió alap formátumának jelzőjét.
hex	Beállítja a hexadecimális konverzió alap formátumának jelzőjét.
oct	Beállítja az oktális konverzió alap formátumának jelzőjét.
ws	Kiszűri a *whitespace* karaktereket.
endl	Beszúr egy új sort.
ends	A sztringet nullal terminálja.
flush	Frissíti az outputot.
setbase(int n)	Beállítja a konverzió alapját az *n* változóval (0,8,10 vagy 16). 0 az alapértelmezés: decimális output.
resetiosflag(long f)	Törli az *f* által megadott formátumbiteket.
setiosflag (long f)	Beállítja az *f* által specifikált formátum bitjeit.
setfill(int c)	Beállítja a *c* paraméterből a töltő karaktert.
setprecision(int n)	Beállítja a lebegőpontos szám pontosságát az *n*-ben.
setw(int n)	Beállítja a mezőszélességet az *n* által.

Bármikor frissíthetjük az *ostream* adatfolyamot a

```
ostream << flush;
```

utasítással.

A változókat a ***printf*** függvényhez hasonlóan fixpontos, lebegőpontos, stb formátumokban írathatjuk ki, különféle jelzőbitek beállításával.

Az 1.12. táblázat a formátum jelzők alapértelmezés szerinti beállítását foglalja össze.

1.12. táblázat

Jelző	Jelentése	Alapértelmezés
`ios::fixed`	Kiírás fixpontosan történik, ilyenkor ki van kapcsolva az ios::scientific jelző.	nincs beállítva
`ios::scientific`	E hatványkitevővel normál alakban történik a kiíratás, ilyenkor ki van kapcsolva az ios::fixed.jelző. Ha sem a fixed, illetve sem a scientific jelző nincs bekapcsolva, akkor az output az eredmény nagyságrendjétől függően fix vagy lebegőpontos alakú lesz.	nincs beállítva
`ios::showpoint`	A tizedespont és a tizedespont utáni vezető nullák is kiírásra kerülnek, ellenkező esetben viszont nem.	nincs beállítva
`ios::showpos`	A jelző beállításakor a + (pozitív előjel) kiíródik a pozitív egész szám előtt.	nincs beállítva
`ios::right`	Ha ez a jelző be van állítva, akkor a megadott mezőszélesség esetén az output jobbra tömörítve jelenik meg. Ilyenkor az ios::left jelző ki van kapcsolva.	beállítva
`ios::left`	Beállítása esetén, a megadott mezőszélesség esetén az output balra tömörítve jelenik meg, ilyenkor az ios::right jelző ki van kapcsolva.	nincs beállítva

1.8.2. Szabványos input

A szabványos inputnál az outputhoz hasonlóan a felülbírált >> jobbra shift operátort használjuk.

```
cin >> adat;
```

A *cin* a C-ben szokásos *stdin* megfelelője.

A **signed** vagy **unsigned char** típusnál a >> operátor átlépi a *whitespace* karaktereket (szóköz, tab, újsor stb.), csak a nem *whitespace* karaktert olvassa be. Ha szükségünk van az aktuális következő karakterre, akkor inkább a ***get*** tagfüggvényt használjuk.

1.8.3. A hibajelzés használata

Létezik még egy szabványos hiba *stream* is, a ***cerr*** (A C-ben szokásos *stderr* megfelelője).

```
cerr << "olvasási hiba";
```

1.8.4. A szabványos I/O bemutatása példákon keresztül

Az eredményt fixpontosan 6 tizedesre írja ki.

```
/* io2.cpp */
#include <iostream.h>
void main(void)
{
 double r; const double pi = 3.141592;
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
}
```

A program futásának eredménye:

```
A kör sugara: 2.5
A kör területe: 19.634950
```

Az eredményt mindenképpen fixpontosan írja ki, melyet a *fixed* jelző beállításával érünk el.

```
/* io3_1.cpp */
#include <iostream.h>
void main()
{
 double r; const double pi = 3.141592;
  cout.setf(ios::fixed);
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
  cout << "A kör kerülete: " << 2*r*pi << endl;
}
```

A program futásának eredményei:

```
A kör sugara: 2.5
A kör területe: 19.63495
A kör kerülete: 15.70796
```

```
A kör sugara: 3000
A kör területe: 28274328
A kör kerülete: 18849.552
```

Az eredményt mindenképpen fixpontosan írja ki, melyet a *fixed* jelző beállításával érünk el. Használjuk a *showpoint* jelzőt is!

```
/* io3_2.cpp */
#include <iostream.h>
void main()
{
 double r; const double pi = 3.141592;
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
  cout << "A kör kerülete: " << 2*r*pi << endl;
}
```

A program futásának eredményei:

```
A kör sugara: 2.5
A kör területe: 19.634950
A kör kerülete: 15.707960

A kör sugara: 3000
A kör területe: 28274328.000000
A kör kerülete: 18849.552000
```

A *scientific* jelző beállításával lebegőpontos kiírást érünk el.

```
/* io3_3.cpp */
#include <iostream.h>
void main()
{
 double r; const double pi = 3.141592;
  cout.setf(ios::scientific);
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
  cout << "A kör kerülete: " << 2*r*pi << endl;
}
```

A program futásainak eredménye:

```
A kör sugara: 2.5
A kör területe: 1.963495e+01
A kör kerülete: 1.570796e+01
```

```
A kör sugara: 3000
A kör területe: 2.827433e+07
A kör kerülete: 1.884955e+04
```

Beállíthatjuk a tizedek számát a *precision* segítségével.

```
/* io3_4.cpp */
#include <iostream.h>
void main()
{
 double r; const double pi = 3.141592;
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.precision(4);
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
  cout.precision(2);
  cout << "A kör kerülete: " << 2*r*pi << endl;
}
```

A program futásainak eredményei:

```
A kör sugara: 2.5
A kör területe: 19.6349
A kör kerülete: 15.71

A kör sugara: 3000
A kör területe: 28274328.0000
A kör kerülete: 18849.55
```

A *showpos* jelző beállításával a pozitív előjel kiírásra kerül.

```
/* io3_5.cpp */
#include <iostream.h>
void main()
{
 double r; const double pi = 3.141592;
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);
  cout.precision(4);
  cout << "A kör sugara: ";
  cin >> r;
  cout << "A kör területe: " << r*r*pi << endl;
  cout.precision(2);
  cout << "A kör kerülete: " << 2*r*pi << endl;
}
```

A program futásainak eredményei:

```
A kör sugara: 2.5
A kör területe: +19.6349
A kör kerülete: +15.71
A kör sugara: 3000
A kör területe: +28274328.0000
A kör kerülete: +18849.55
```

A kiírás a megadott mezőszélességnek megfelelően jobbra tömörítve jelenik meg.

```
/* io4.cpp */
#include <iostream.h>
#include <iomanip.h>
void main()
{
 int x = -12, y = 251, z = 1200;
  cout << "123456789012345678\n";
  cout << setw(4) << x;
  cout << setw(6) << y;
  cout << setw(8) << z;
}
```

A program futásának eredménye:

```
123456789012345678
 -12   251    1200
```

Lehetőség van az egész számot decimálisként, oktálisan és hexadecimálisan kiíratni a *dec*, *hex* ill. *okt* beállításával.

```
/* io5.cpp */
#include <iostream.h>
#include <iomanip.h>
void main()
{
 int sz = 28;

   cout << dec << "decimális    : " << sz << endl;
   cout << hex << "hexadecimális: " << sz << endl;
   cout << oct << "oktális      : " << sz << endl;
   cout << "\n";
   cout << dec << "decimális    : " << sz << endl;
}
```

A program futásának eredménye:

```
decimális      : 28
hexadecimális: 1c
oktális        : 34

decimális      : 28
```

A *fill* segítségével különféle karakterrel tölthetjük fel az üres helyeket a jobbra illetve balra tömörítésnél. A *right, left* illetve *internal* váltásnál használjuk az *adjustfield* long konstans adatmezőt.

```
/* io6.cpp */
#include <iostream.h>
#include <iomanip.h>
void main()
{
 int sz = 28;
  cout.fill('#');
  cout.width(5);
  cout << dec << sz << endl;
  cout.width(8);
  cout.fill('*');
  cout.setf(ios::left,ios::adjustfield);
  cout <<  hex << sz << " ";
}
```

A program futásának eredménye:

```
###28
1c******
```

A *cin* adatfolyammal különféle típusokat olvashatunk be.

```
/* io7.cpp */
#include <iostream.h>
#include <iomanip.h>
void main()
{
 char   c;
 int    k;
 float  f;
 double d;
   cin >> c >> k >> f >> d;
   cout << " c: " << c;
   cout << "  k: " << k;
   cout << "  f: " << f;
   cout << "  d: " << d;
}
```

A program futásának eredménye:

```
c: *  k: 5  f: 12.1  d: 4.5674
```

1.8.5. Az unset használata

A *setf* segítségével bekapcsoljuk a + előjel kiíratását:

```
cout.setf(ios::showpos);
```

Az *unsetf* pedig a jelzőt kikapcsolja. Például:

```
cout.unsetf(ios::showpos);
```

1.8.6. A get és a put

Az input adatfolyamnak a ***get*** tagfüggvényével egy karaktert olvashatunk a szabványos bemenetről. A ***get*** olvassa a következő karaktert, amely szóköz és újsor is lehet és karakter típusú változóba helyezi.

```
input_stream.get(kar_valtozo);
```

Az output adatfolyamnak a ***put*** tagfüggvényével egyetlen karaktert írhatunk ki a szabványos kimenetre.

```
output_stream.put(karakter_kifejezes);
```

Néha a programnak szükséges, hogy mi a következő karakter az input adatfolyamban. Ilyenkor beolvassuk a következő karaktert, megvizsgálhatjuk és ha nem a várt karaktert kaptuk, akkor lehetőség van a ***putback*** tagfüggvénnyel az input adatfolyamba való visszahelyezésre.

1.8.6.1. Példaprogramok a get és a put használatára

Egy karakter olvasása a *get* tagfüggvénnyel, illetve egy karakter írása a *put* tagfüggvénnyel.

```
/* ch_io1.cpp */
#include <iostream.h>

void main()
{
  char ch1,ch2,ch3,ch4,ch5 ;
  cout << "\n5 karakteres szót olvas: ";
  cin.get(ch1); cin.get(ch2);
  cin.get(ch3);cin.get(ch4);cin.get(ch5);
```

```
  cout << "\nBeolvasott karakterek: " << ch1
       <<ch2 << ch3 << ch4 << ch5 <<"\n";
  cout << "Más sorrendben kiírva: ";
  cout.put(ch3); cout.put(ch4); cout.put(ch2);
  cout.put(ch1); cout.put(ch5);
}
```

A program futásának eredménye:

```
5 karakteres szót olvas: kocsi

Beolvasott karakterek: kocsi
Más sorrendben kiírva: csoki
```

A * után karaktert, # után egész számot és & után valós számot olvasunk.

```
/* ch_io3.cpp */
#include <iostream.h>

void main()
{
 char   ch1 ,ch2;
 int    szam;
 double dszam;

  cout << "karakter (*,#,&): ";
  cin.get(ch1);
  switch (ch1)
  {
   case '*': cin.get(ch2); cout << "Karakter: " << ch2; break;
   case '#': cin >> szam;  cout << "Egész szám: " << szam; break;
   case '&': cin >> dszam; cout << "Duplapontos szám: "
                     << dszam; break;
  }
}
```

A program futásainak eredménye:

```
karakter (*,#,&): *$
Karakter: $

karakter (*,#,&): # 12
Egész szám: 12

karakter (*,#,&): & 12.56

Duplapontos szám: 12.56
```

1.9. Kifejezések

A kifejezések önmagukban utasítás értékűek is lehetnek, mert a C(++) nyelvi kifejezés jóval tágabb értelmű, mint a más nyelvekben megszokott. A kifejezések formailag vagy elsődleges kifejezések lehetnek, vagy részkifejezés(ek)ből épülnek fel operátor(ok) segítségével. A C(++) nyelvben létezik egyoperandusú (*unary*), kétoperandusú (*binary*) és háromoperandusú (*ternary*) operator.

Egyoperandusú pl. a negálás operátora: -x
Kétoperandusúra példa a szorzás operátora: x*y
A makróként használt feltételes operátor viszont: x<y ? x : y

1.9.1. Elsődleges kifejezések

Elsődleges kifejezés az azonosító - feltéve, hogy már teljes körűen deklaráltuk. Típusa a deklarációnak megfelelő. Elsődleges kifejezés továbbá minden konstans, az ott tárgyalt megfelelő típussal, beleértve a sztringkonstansokat is, amelyek típusa karaktertömb. Bármely kifejezés ( ) zárójelek közé zárva szintén elsődleges kifejezéssé válik, örökölve az adott kifejezés típusát.

Elsődleges kifejezés az őt közvetlenül követő [ ] zárójelek közt álló kifejezéssel együtt elsődleges kifejezést alkot. Más szóval, egy tömbváltozót indexelve szintén elsődleges kifejezést kapunk, aminek a típusa az adott tömb alaptípusa lesz. Ebben az összefüggésben a [ ] a benne szereplő indexelő kifejezéssel együtt az ún. indexelő operátor, amelynek használata formailag hasonlít az 1.7.5. pontban említett [ ] tömbtípust képző operátor megjelenésére, de míg ez utóbbi a fordító programot arra utasítja, hogy az alaptípusból tömbtípust hozzon létre, addig az indexelő operátorral egy tömbtípusú tárolási egység egy, az alaptípusba tartozó elemét jelöljük ki.

A függvényhívás is elsődleges kifejezés, ami formailag egy elsődleges kifejezés az őt közvetlenül követő ( ) zárójelek közé zárt, vesszővel elválasztott kifejezéslistával (aktuális paraméterlistával) együtt. A függvényhívás eredményének típusa az adott függvény deklarált visszatérő típusa. A tömbindexeléshez hasonlóan úgy is felfoghatjuk a dolgot, hogy a *postfix* ( ) függvényaktivizáló operátort alkalmazzuk az adott függvénytípusú tárolási egységre, és ennek a műveletnek az eredménye az adott függvénytípus deklarációja szerinti alaptípus, azaz a függvény visszatérési értékének a típusa lesz. (Itt megint csak a 1.7.5. pontban elmondottakra utalunk.) Ez a gondolkodásmód összhangban van

azzal, hogy egy azonosító mindig elsődleges kifejezés. A függvényhívás is elsődleges kifejezés: önmagában leírva definíció szerint a függvény címét jelenti. Ezzel mást nem lehet csinálni, mint értékül adni egy, az adott típusú függvényre mutató pointernek, vagy a függvényaktivizáló operátort alkalmazni rá, miáltal az adott függvényben programkód az aktuális függvényparaméterekkel lefut.

Elsődleges kifejezések továbbá a struktúrák és unionok mezőire vonatkozó kiválasztó operátorok (a . és a ->) alkalmazásával nyert olyan kifejezés, amelyeknél a operátor bal oldalán elsődleges kifejezés, jobb oldalán pedig azonosító áll. Ezeket később részletesen fogjuk ismertetni.

1.9.2. Operátorok

A C(++) nyelv egyik erőssége más nyelvekhez képest a kifejezésekben használható operátorok gazdag választéka. Az eddig említett operátorokat (típusmódosító operátorok, indexelő, illetve függvényaktivizáló operátorok) csak a C(++)-ben nevezik operátoroknak.

A következőkben a hagyományos értelemben vett operátorokat ismertetjük. Ezek többnyire elemi típusú adatokon végeznek műveleteket, de egyik-másik közülük származtatott, illetve bonyolultabb, összetett típusú tárolási egységekre is alkalmazható. Az egyes operátorok felhasználásának lehetőségét a C++ az objektum-orientált tulajdonságai révén még tovább szélesíti. Ezt majd az ún. operátor *overloading* kapcsán tekintjük át részletesebben.

A következőkben tehát leírjuk a C(++) nyelvben hagyományos értelemben vett operátorokat, ismertetjük azok alkalmazását.

Az operátoroknak különböző precedenciájuk lehet, a kiértékelés sorrendje ezektől függ. A fejezet végén ezért majd összefoglalóan felsoroljuk az egyes precedencia-osztályokat, és megadjuk azt is, hogy mi a teendő, ha azonos precedenciájú operátorok együtt fordulnak elő. A legtöbb kétoperandusú operátor megkívánja, hogy mindkét operandusa azonos típusú legyen. Például a /-rel osztási művelet mást jelent egész típusú operadunsokra alkalmazva (maradékos egész osztás), mint lebegőpontosakra. Ha a műveletben résztvevő két operandus nem azonos típusú, akkor az egyik átalakul, hogy megegyezzenek, mindig a sorszámozott típusú alakulva lebegőpontossá, a rövidebb változat a hosszabbra, az előjeles az előjel nélkülire.

1.9.2.1. Egyoperandusú operátorok

Az egyoperandusú operátorok közé tartozik a [] indexelő, a () függvényaktivizáló operátor és az elsődleges kifejezést képző operátor, az egyszerű zárójel pár. Ezeket az 1.7.5. pontban ismertettük, itt csak a teljesség kedvéért említjük meg őket újra.

Az egyoperandusú * operátor, az ún. *dereference operator* indirekciót jelez, az operandusa mutató kell, hogy legyen, eredményül a mutató által megcímzett értéket kapjuk.

Az indirekció operátor inverze az egyoperandusú & operátor, amely az operandusa címét szolgáltatja (*'address of' operator*).

Az egyoperandusú - (mínusz) operátor az operandus 2-es komplemensét szolgáltatja, a szokásos aritmetikai konverziók elvégzése után. Előjel nélküli egészekre alkalmazva ezt az operátort, az eredményre igaz lesz, hogy hozzáadva az eredeti operandust az összeg 2^n, ahol n az operandus szélessége bitekben.

A logikai tagadás a ! (felkiáltójel), eredménye int típusú 1 vagy 0, attól függően, hogy az operandus 0 volt-e vagy sem. Alkalmazható lebegőpontos számokra és mutatókra is.

A bitenkénti komplementálás operátora a ~ (a tilde karakter) , ami a - kötelezően sorszámozott típusú - operandusán elvégzett szokásos aritmetikai konverziók után kapott bitminta 1-es komplemensét adja eredményül.

Az előtagként alkalmazott (prefix) egyoperandusú ++ operátor operandusának az értékét megnöveli 1-gyel, és eredményül ezt a megnövelt értéket szolgáltatja olyan típusban, mint amilyen az operandusé. Ez az operátor tehát nemcsak visszaad egy értéket, de mellékhatása is van az operandusára.

```
x = 3;
y = ++x*2;
```

Ha *x* értéke 3, a ++*x* hatására *x* felveszi a 4 értéket, az *y* pedig 8 lesz.
Analóg módon létezik egyoperandusú prefixdekrementáló operátor is, jele --.

Ez operandusa értékét csökkenti 1-gyel, és ezt adja értékül.

```
x = 3;
y = --x*2;
```

Ha *x* értéke 3, a *--x* hatására *x* felveszi a 2 értéket, az *y* pedig 4 lesz.

Utótagként (postfix) is alkalmazható az egyoperandusú ++ operátor. Ekkor operandusának az értékét 1-gyel megnöveli, de eredményül a növelés előtti értékét szolgáltatja olyan típusban, mint amilyen az operandusé. Ennek az operátornak tehát szintén mellékhatása van az operandusára, de az eredményben ez nem jelentkezik.

```
x = 3;
y = x++ * 2;
```

Ha *x* értéke 3, a *x++* hatására *x* felveszi a 4 értéket, az *y* pedig 6 lesz.
Analóg módon létezik egyoperandusú postfix dekrementáló operátor is, jele --. Ez operandusa értékét csökkenti 1-gyel, és eredményül a csökkentés előtti értéket szolgáltatja.

```
x = 3;
y = x-- * 2;
```

Ha *x* értéke 3, a *x--* hatására *x* felveszi a 2 értéket, az *y* pedig 6 lesz.

A típuskonverzió (*type cast*) operátora az adott operandus előtt zárójelben álló típusnév (absztrakt deklarátor). Az eredmény értéke - a lehetőségek határain belül - megegyezik az operandusével, típusa a megnevezett új típus.
Például:

`(int)3.141`	értéke 3,
`(long)0`	értéke megegyezik a 0L-val,
`(long*)p`	olyan mutatót eredményez, ami ugyanarra a tárterületre mutat, mint p, de a megcímzett memóriarészt a fordító hosszú egésznek tekinti.

Ez utóbbi talán a legjellemzőbb a típuskonverzió operátorának alkalmazására. Így alakítjuk át az általános - ismeretlen típusú változóra mutató - *void** "típusú" mutatókat adott típusú mutatókká.

A ***sizeof*** egyoperandusú operátor az operandusaként szereplő változó (vagy zárójelben álló típus) *byte*-okban megadott méretét szolgáltatja. Ennek segítségével lehet gépfüggetlen módon tárterület igényeket meghatározni.

Itt jegyezzük meg, hogy mind a C, mind a C++ nyelv definíciója csak annyit közöl az egyes elemi típusok méreteiről, hogy azok minden implementációban meg kell hogy feleljenek az alábbi relációknak:

```
1 ≡ sizeof(char) ≤ sizeof(short) ≤ sizeof(int) ≤ sizeof (long),
```

illetve

```
sizeof(float) ≤ sizeof(double).
```

A konkrét méretek a nyelv adott implementációjától függnek. A *sizeof(**int**)* 16 bites Borland C++ fordító esetén 2, 32 bites fordító esetén pedig 4. A fenti példákból is látszik, hogy az implementáció-független, portábilis C, illetve C++ programozási stílus kialakításában a ***sizeof*** operátornak kulcsszerepe van.

1.9.2.2. Kétoperandusú operátorok

Aritmetikai műveleti jelek:

+ összeadás
- kivonás
* szorzás
/ osztás
% maradékképzés

A kétoperandusú + és - operátorok a szokásos összeadást, illetve kivonást végzik el , a szokásos aritmetikai konverziók után. Speciális esetként lehetnek mutató operandusok is.

A kétoperandusú * operátor a szorzás művelete, a / operátor pedig az osztásé. A szokásos aritmetikai konverziók itt is a műveletek elvégzése előtt történnek meg. A maradékképzés operátora a %, amelynek operandusai kötelezően sorszámozott típusúak.

Egészek osztásánál, ha bármelyik operandus negatív, a csonkítás iránya gépfüggő, és hasonlóképpen gépfüggő az is, hogy a maradék az osztó vagy az osztandó előjelét örökli-e. Pozitív egészek osztásánál mindig lefelé csonkítás történik. Minden esetben fennáll azonban, hogy *(a/b)*b+a%b* megegyezik *a*-val, ha *b* nem nulla.

Biteltoló operátorok

<<	eltolás balra
>>	eltolás jobbra

A << és >> kétoperandusú operátorok bitenkénti eltolás (*shift*) műveletet végeznek balra, illetve jobbra. Mindkét operandusnak sorszámozott típusúnak kell lenni, és a szokásos aritmetikai konverziók után az eredmény típusa a bal oldali operandusával egyezik meg. A kilépő bitek elvesznek, balra léptetésnél az üres helyekre 0-ák lépnek be. Jobbra történő eltolásnál a belépő bitek garantáltan 0-ák, ha a bal oldali operandus előjel nélküli értelmezésű, egyébként értékük gépfüggő.

A Borland C++ esetében az előjeles jobbra léptetésnél a belépő bitek az eredeti érték előjelbitjével egyeznek meg.
Például

2 << 3	értéke 16,
0xFFFDu>>2	eredménye 0x3FFFu, míg
-3 >> 1	értéke -1.

A relációs és egyenlőség vizsgáló operátorok a következőek:

<	kisebb,
>	nagyobb,
<=	kisebb vagy egyenlő,
>=	nagyobb vagy egyenlő
==	egyenlő
!=	nem egyenlő

Értelmezésük a szokásos, eredményül int 1-et vagy 0-át adnak, attól függően, hogy a vizsgált feltétel teljesül-e vagy sem.

Bitenkénti operátorok

&	bitenkénti ÉS
^	kizáró VAGY
|	bitenkénti VAGY

A bitenkénti operátorok sorszámozott típusú operandusaikon végeznek a szokásos aritmetikai konverziók után bitműveleteket. Ezek a bitenkénti ÉS (AND) operátor: kétoperandusú &, a bitenkénti KIZÁRÓ VAGY (XOR) operátor: ^ és a bitenkénti VAGY (OR) operátor: |.

Logikai operátorok

&& logikai ÉS
|| logikai VAGY

A logikai ÉS operátor (&&) és a logikai VAGY (||) logikai értékű operandusokat vár (0 - hamis, nem 0 - igaz).

1.13. táblázat

a	b	a &&b
igaz	igaz	igaz
igaz	hamis	hamis
hamis	igaz	hamis
hamis	hamis	hamis

<table>
<tr><th>a</th><th>b</th><th>a ||b</th></tr>
<tr><td>igaz</td><td>igaz</td><td>igaz</td></tr>
<tr><td>igaz</td><td>hamis</td><td>igaz</td></tr>
<tr><td>hamis</td><td>igaz</td><td>igaz</td></tr>
<tr><td>hamis</td><td>hamis</td><td>hamis</td></tr>
</table>

Eredményül **int** 0-át vagy 1-et szolgáltatnak a megfelelő logikai művelet elvégzése után.

Háromoperandusú operátor

A feltételes (? :) operátor az egyetlen háromoperandusú operátor a C nyelvben, alakja:

kif1 ? kif2 : kif3;

A kifejezés feloldása a *kif1* kifejezés kiértékelésével kezdődik. Ha az eredmény nem 0, akkor a *kif2*, egyébként pedig a *kif3* kifejezés értékelődik ki, és ez utóbbi érték adja a művelet eredményét és típusát. *kif2*-nek és *kif3*-nak a szoká-

sos aritmetikai konverziók után azonos típusúaknak kell lenniük, kivéve, ha az egyik mutató, a másik pedig konstans 0. Ekkor az eredmény típusa a mutató típusa lesz. Garantált, hogy futás közben *kif2* és *kif3* közül csak az egyik kerül kiértékelésre.

Értékadás operátora

A C(++) nyelvben explicit értékadó utasítás nincs; az értékadás operátorokon keresztül valósul meg, kifejezések kiértékelésének mellékhatásaként. A leggyakrabban használt értékadó operátor az egyszerű értékadás operátora, jele =. Például

```
a = b+3*c;
```

Kiértékelésre kerül a jobb oldali kifejezés (szükség szerint konvertálva a bal oldal típusának megfelelően), majd beíródik a bal oldalon meghatározott változókba, felülírva annak korábbi tartalmát. Az egyszerű értékadó operátor bal oldalán álló kifejezést az angol terminológia alapján balérték (*lvalue*), a jobb oldalán álló kifejezést pedig jobbérték (*rvalue*) nevezzük. A kifejezés eredménye és típusa az értékadás során átadott értéknek megfelelő.

További értékadó operátorok:

+=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |= .

Látható, hogy mindegyik egy kétoperandusú operátorból és az értékadás jeléből tevődik össze. Egy

```
kif1 op = kif2
```

formájú kifejezés kiértékelését úgy tekintjük, mintha a helyén

```
kif1 = kif1 op kif2
```

alakú értékadás állna - ahol *op* a fenti kétoperandusú operátorok bármelyike lehet. Fontos különbség azonban, hogy *kif1* kiértékelésére csak egyszer kerül sor. Például ha

```
x = 1;
x +=2;
```

akkor az *x+=2* kifejezés értéke 3 és mellékhatásként *x* is 3 lesz.

Vessző operátor

A vesszőoperátor (comma operator) nevét jeléről, a , karakterről kapta. A vesszőoperátorral elválasztott kifejezés sorban egymás után kiértékelődik, és az eredmény értéke és típusa megegyezik a második (jobb oldal) kifejezésével. Ha a vessző egy adott kontextusban másképp is előfordulhat (függvény paramétereinek elválasztásánál, kezdeti értékek listájánál), akkor a vesszőoperátorral felépített kifejezést zárójelekkel kell védeni, például:

```
fugg(a, (t = 3, t+2), c);
```

A fenti függvénynek három paramétert adunk át, amely közül a középső értéke 5.

Az előzőekben felsorolt operátorok előfordulási sorrendje megegyezik a prioritásuk sorrendjével. Az 1.14. táblázatban az egyes prioritási szintek csökkenő sorrendben a hozzájuk tartozó associativitási iránnyal együtt láthatóak.

1.14. táblázat

Elsődleges kifejezések	(), [] . ->	Az elsődleges kifejezések balról jobbra csoportosítanak, tehát a a.k[3] értelmezése (a.k)[3]. Itt a . (a pont karakter) az ún. mezőkiválasztó operátor: erről részletesebben a struktúrák ismertetésénél lesz szó.
Egyoperandusú operátorok	*,&,-,+,!,~,++,-- típusmódosítás sizeof	Végrehajtás balról jobbra.
Multiplikatív operátorok	*,/,%	Végrehajtás balról jobbra.
Additív operátorok	+, -	Végrehajtás balról jobbra.
Biteltoló operátorok	<<, >>	Végrehajtás balról jobbra.
Relációs operátorok	<,<=	Végrehajtás balról jobbra.
Egyenlőség-vizsgáló operátorok	= =, !=	Végrehajtás balról jobbra.
Bitenkénti ÉS operátor	&	Megjegyzés:1. és 3.
Bitenkénti VAGY, kizáróVAGY operátorok.	|	Megjegyzés: 1.
Logikai ÉS operátor	&&	Megjegyzés: 2.
Logikai VAGY, kizáró VAGY operátorok	||	Megjegyzés: 2.

Megjegyzések

1. A kétoperandusú: *,+,&,- és az | operátorok asszociatívak. Ha egy kifejezésben azonos szinten több azonos asszociatív operátor szerepel, akkor a fordítónak jogában áll a kiértékelési sorrendet tetszőlegesen megváltoztatni. Ez - a részkifejezések mellékhatásai miatt - kihathat az eredmény értékére is, ezért kerülendők az olyan kifejezések, amelyek nem definit kiértékelési sorrendtől függenek. Például az

   ```
   a++ + b[a]
   ```

   kifejezésben a *b* tömb indexe attól függ, hogy az első vagy második tag kiértékelése történik előbb, azaz az indexelésben az eredeti vagy a megnövelt *a*-t használjuk. Ugyanez a helyzet áll fenn a függvényhívások paramétereinél, ugyanis ezek kiértékelése sem kötött sorrendű. Például:

   ```
   fugg(i++, a[i])
   ```

   gépfüggően választja meg adott *i* érték mellett az átadandó tömbelemet.
2. A logikai operátorok *(&&,* ||) garantálják a balról jobbra történő kiértékelési sorrendet. Ezenkívül azt is biztosítják, hogy a második kifejezés kiértékelését csak akkor végzik el, ha az első operandusok alapján az eredmény nem egyértelmű. Például *a && b* esetében *b* kiértékelésre nem kerül sor, ha az a 0, hiszen, ekkor az eredmény már biztosan hamis logikai érték lesz. Ennek jelentőségére igyekszik rávilágítani a következő feltétel:

   ```
   b != 0 && a/b <5
   ```

   Ha nem lenne garantált a fenti kiértékelési stratégia, akkor ez a feltétel hibás lenne, mert *b* 0 értéke esetén bekövetkezne az elkerülni kívánt 0-val történő osztás. Ügyeljünk arra, hogy a logikai kifejezésekben fellépő mellékhatásokat ne használjuk ki, ugyanis egy logikai kifejezés esetleg lerövidült kiértékelése következtében a várt mellékhatások egy része elmaradhat.
3. A bitenkénti operátorok precedenciája alacsonyabb, mint az egyenlőség-vizsgáló operátoroké. Gyakori hiba a következő alakú vizsgálat:

   ```
   c & 0xF = = 8
   ```

   Ez garantáltan mindig 0-át ad eredményül. Helyes megoldás:

   ```
   (c & 0xF) = = 8
   ```

4. A feltételes operátor *kif* feltételrészében és minden egyéb helyen, ahol feltételt kell megadnunk, tetszőleges kifejezés szerepelhet és annak 0, illetve nem 0 értéke jelenti a feltétel hamis, illetve igaz voltát. Különösen veszélyes ez akkor, ha tévedésből összekeverjük az egyszerű értékadó operátor = jelét az egyenlőség-vizsgáló operátor = = jelével.
   Az $a = b$ alakú feltétel ugyanis szintaktikailag teljesen helyes, de nem a két mennyiség egyenlőségét vizsgálja, hanem *b 0* vagy nem *0* volta szerint szolgáltatja az eredő feltételt, egyben *b* értékével *a*-t is felülírva. Szerencsére a Borland C++ az ilyen feltételeknél - ha engedélyezzük - figyelmeztetést ad (a *Possibly incorrect assigment* üzenettel). Ha viszont ténylegesen a fenti esetre van szükségünk, akkor a figyelmeztetés elkerülése - és az érthetőbb felírási forma - kedvéért írjuk azt az

   ```
   (a = b) != 0
   ```

   alakban.

1.10. Konverziók

A Borland C++ a standard eljárásokat alkalmazza az egyes adattípusok közötti automatikus konverziókra. A következőkben az implementációfüggő részeket ill. az alkalmazott bővítéseket ismertetjük.

1.10.1. Aritmetikai konverziók

Ha egy aritmetikai kifejezés, például $a+b$ esetén, ahol az a és b operandusok különböző típusúak, akkor a Borland C++ bizonyos belső konverziókat hajt végre, mielőtt a kifejezést kiértékelné.

Nézzük meg, hogy a Borland C++ hogyan konvertálja az oprandusokat az aritmetikai kifejezésekben:

1. A nem egész, az **int**-nél rövidebb értékek egésszé, a szimpla pontosságú lebegőpontos **double** típusúvá kovertálódnak. E lépés után mindkét operandus **int**, illetve **double** lesz, beleértve a **long** illetve **unsigned** módosító jelzőket.
2. Ha az egyik operandus **long double**, a másik operandus is **long double** típusúvá alakul át.
3. Egyébként, ha bármelyik operandus **double**, akkor a másik operandus is **double** típusúvá alakul át.

4. Egyébként, ha bármelyik operandus **float**, akkor a másik operandus is **float** típusúvá alakul át.
5. Egyébként, ha bármelyik operandus **unsigned long**, akkor a másik operandus is **unsigned long** típusúvá alakul át.
6. Egyébként, ha bármelyik operandus **long** típusú, akkor a másik operandus is **long** lesz.
7. Egyébként, ha bármelyik operandus **unsigned**, akkor a másik operandus is **unsigned** lesz.
8. Máskülönben mindkét operandus típusa **int**.

A kifejezés eredményének típusa minden esetben megegyezik a két operandus típusával.

1.15. Táblázat

Típus	Mire konvertálódik	módszer
char	**int**	Nulla vagy előjel-kiterjesztés.
unsigned int	**int**	Nulla értékű felső bájttal bővítve.
signed char	**int**	Mindig előjel-kiterjesztés.
short	**int**	Előjelesen ugyanaz az érték másolva.
unsigned sort	**unsigned int**	Ugyanaz az érték nullával kitöltve.
enum	**int**	Ugyanaz az érték másolva.

1.10.2. Konverzió a char, az int és az enum típusok között

Karakterállandót egész típusú objektumnak értékül adva teljes 16 bites értékadást végez a gép, mivel minden karakterállandó 16 bites mennyiség. Egy **char** típusú tárolási egység sorszámozott mennyiséghez való hozzárendelése esetén előjel-kiterjesztés történik. A **signed char** típusú tárolási egységekre mindig előjel-kiterjesztés történik, az **unsigned char** típusúak pedig mindig 0 értékű felső bájttal lesznek 16 bitre bővítve.

Konvertálva **long** típusról **short** típusra a felső bájt elveszik, míg az alsó bájt változatlan marad. Konvertálva **short** típusról **long** típusra, ebben az esetben a felső bájt 0-val lesz kitöltve aszerint, hogy a **short** típus **singed** vagy **unsigned** volt.

Az **enum** és **int** értékek közötti konverzió egyszerű értékmásolással történik, az **enum** értékek és a karakterek konverziójára ugyanaz a szabály érvényes, mint az **int** értékek és a karakterek esetében.

1.11. Utasítások

A C++ nyelvi utasítások csak valamely kódgeneráló tárolási egység definíciójánál, a függvénytörzset alkotó blokkban fordulhatnak elő. Az utasítások végrehajtása leírásuk sorrendjében történik, kivéve a vezérlésátadó utasításokat.

1.11.1. Kifejezés-utasítások

Bármely kifejezés pontosvesszővel lezárva szintén utasítás, és a kifejezés kiértékelését vonja maga után. A kifejezés lehet függvényhívás, értékadás. Az értékadás a = (egyenlőség) jellel történik. Kifejezés-utasítások azok is, amelyek inkrementáló, vagy dekrementáló operátort tartalmaznak, például x++; Ezekben az esetekben lényegtelen, hogy a prefix vagy a postfix változatot használjuk.

Nézzünk meg néhány feladatot az értékadás, a ++, -- operátorok és a ***sizeof*** használatára.
A prg_2_1.cpp programban megfigyelhetjük
- az egész típusú változó a valós változó tartalmát egészre vágva veszi át hibajelzés nélkül,
- két egész változó osztásakor egész lesz az eredmény,
- ha az egész típusú osztó vagy az egész típusú osztandó **double** típusra van átalakítva, akkor az eredmény valós lesz.

```
/* prg_2_1.cpp */
#include <iostream.h>
void main()
{
  int i1 = 12, i2;
  long k1 = 198, k2;
  float a1 = 3.567, a2;
  double b1 = 21.89767, b2, b3, b4;
   i2 =a1;
   a2 = k1;
   k2 = k1/i1;
   b2 = k1/i1;
```

```
    b3 = (double)k1/i1;
    b4 = k1/(double)i1;
    cout << "i1 = " << i1 << endl;
    cout << "i2 = " << i2 << endl;
    cout << "k1 = " << k1 << endl;
    cout << "k2 = " << k2 << endl;
    cout << "a1 = " << a1 << endl;
    cout << "a2 = " << a2 << endl;
    cout << "b1 = " << b1 << endl;
    cout << "b2 = " << b2 << endl;
    cout << "b3 = " << b3 << endl;
    cout << "b4 = " << b4 << endl;
}
```

A program futásának eredménye:

```
i1 = 12
i2 = 3
k1 = 198
k2 = 16
a1 = 3.567
a2 = 198
b1 = 21.89767
b2 = 16
b3 = 16.5
b4 = 16.5
```

Az egész típusú változók növelésére a ++, és csökkentésére a -- operátort használhatjuk. Ezek az operátorok a változó előtt és után is szerepelhetnek:

- Ha a ++ (ill. --) jelet a változó neve után tesszük, akkor a változó tartalmának a kiolvasása után kerül sor a változó tartalmának a növelésére ill. a csökkentésére.
- Ha a ++ (ill. --) jelet a változó neve elé tesszük, akkor először a változó tartalma kerül növelésre, ezért a kiolvasás már az eggyel nagyobb (ill. kisebb) tartalmat látja.

A prg_2_2.cpp program bemutatja a ++ és a -- alkalmazását.

```
/* prg_2_2.cpp */
#include <iostream.h>
void main()
{
 int i = 5, j = 11, m1;
   m1 = i + j;     cout << "m1 = " << m1 << endl;
   m1 = i++ +j;    cout << "m1 = " << m1 << endl;
   m1 = ++i + j;   cout << "m1 = " << m1 << endl;
   m1 = i-- + j;   cout << "m1 = " << m1 << endl;
   m1 = i + j--;   cout << "m1 = " << m1 << endl;
   m1 = --i + --j; cout << "m1 = " << m1 << endl;
}
```

A program futásának eredménye:

```
m1 = 16
m1 = 16
m1 = 18
m1 = 18
m1 = 17
m1 = 14
```

A ***sizeof*** függvénnyel különféle típusok helyfoglalását határozhatjuk meg.

```
/* prg_2_3.cpp */
#include <iostream.h>
void main()
{
   cout << "char     mérete: "  << sizeof(char) << endl;
   cout << "unsigned mérete: "  << sizeof(unsigned char) << endl;
   cout << "short    mérete: "  << sizeof(short) << endl;
   cout << "int      mérete: "  << sizeof(int) << endl;
   cout << "long     mérete: "  << sizeof(long) << endl;
   cout << "float    mérete: "  << sizeof(float) << endl;
   cout << "double   mérete: "  << sizeof(double) << endl;
}
```

A program futásának eredménye:

```
char     mérete: 1
unsigned mérete: 1
short    mérete: 2
int      mérete: 2
long     mérete: 4
float    mérete: 4
double   mérete: 8
```

A 32 bites fordító esetén az **int** mérete 4 bájtos.

1.11.2. Feltételes utasítás

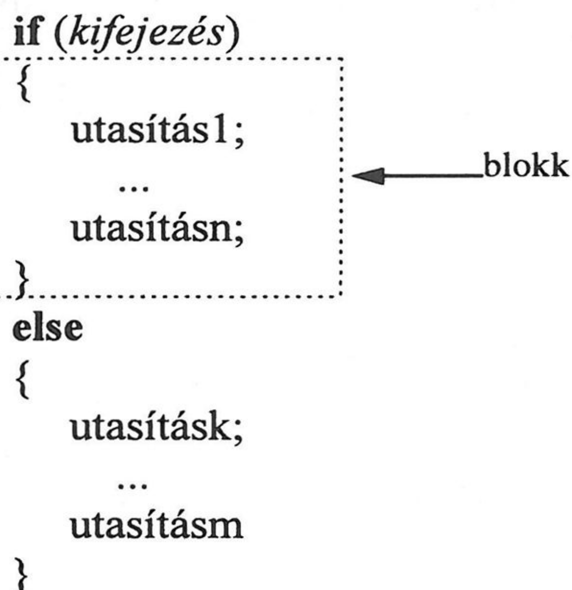
A feltételes utasításnak két formája van.

if (*kifejezés*) utasítás

vagy

if (*kifejezés*) utasítás1;
else utasítás2;

Ha az **if**, valamint az **else** ágban egynél több utasítás van, akkor utasítás zárójelet: {} kell alkalmazni:



A feltételes utasítás a kifejezés kiértékelésével kezdődik, és ha az igaz (nem nulla) , akkor az első utasítás vagy utasításokból álló blokk kerül végrehajtásra. A második formánál megadott második utasításra vagy blokkra akkor kerül a vezérlés, ha a kifejezés hamis (0 értékű).

A kifejezés lehet:
- reláció,
- logikai kifejezés,
- aritmetikai kifejezés,
- változó.

Ha a változó, vagy aritmetikai kifejezés értéke zérus, a feltétel hamis, ha nem zérus, a feltétel igaz lesz.

Nézzünk néhány példát a feltételes utasítás használatára!

Olvassunk be két számot és jelezzük, hogy melyik szám nagyobb a másiknál.

```
/* prg_3.cpp */
#include <iostream.h>
void main()
{
 int a,b;
   cout << "1. adat: "; cin >> a;
   cout << "2. adat: "; cin >> b;
   if ( a > b ) cout << "1. adat > 2. adat";
   else cout << "2. adat > 1. adat";
}
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 10
2. adat > 1. adat
```

A PRG_3.CPP program módosított változata, ahol nemcsak szövegesen írjuk vissza, hogy az adatok közül melyik a nagyobb, hanem az adatokat is visszaírjuk.

```
/* prg_3_1.cpp */
#include <iostream.h>
void main()
{
 int a,b;
   cout << "1. adat: "; cin >> a;
   cout << "2. adat: "; cin >> b;
   if ( a > b )
   {
     cout << "1. adat: " << a;
     cout << " nagyobb, mint a 2. adat: " << b;
   }
   else
   {
     cout << "2. adat: " << b;
     cout << " nagyobb, mint az 1. adat: " << a;
   }
 }
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 10
2. adat: 10 nagyobb, mint az 1. adat: 5
```

A beolvasott két szám vizsgálatánál ne csak a nagyobbat jelezzük, hanem az egyenlőséget is.

```
/* prg_3_2.cpp */
#include <iostream.h>
void main()
{
 int a,b;
   cout << "\n1. adat: "; cin >> a;
   cout << "2. adat: "; cin >> b;
   if ( a > b )
   {
     cout << "1. adat: " << a;
     cout << " nagyobb, mint a 2. adat: " << b;
   }
   else
   if ( b > a)
   {
     cout << "2. adat: " << b;
     cout << " nagyobb, mint az 1. adat: " << a;
   }
   else cout << "a két adat egyenlő.";
 }
```

A program futásának eredménye:

```
1. adat: 5
2. adat: 5
a két adat egyenlô.
```

Olvassuk be a víz hőmérsékletét Celsius fokban és szövegesen írjuk vissza a víz halmazállapotát.

```
/* prg_3_3.cpp */
#include <iostream.h>
void main()
{
 double homerseklet;
   cout << "A víz hőmérséklete: ";
   cin  >> homerseklet;
   cout << "A víz halmazállapota: ";
   if (homerseklet <= 0)  cout << "szilárd";
   else if (homerseklet < 100)  cout << "folyadék";
   else cout << "légnemű";
 }
```

A program futásainak eredménye:

```
A víz hőmérséklete: 50
A víz halmazállapota: folyadék
```

```
A víz hőmérséklete: 100
A víz halmazállapota: légnemű

A víz hőmérséklete: -4
A víz halmazállapota: szilárd
```

1.11.3. Egyéb vezérlésátadó utasítások

A **switch** típusú elágaztatás a vezérlést több lehetséges utasítás valamelyikére adja át egy kifejezés értékétől függően.

switch (*kifejezés*)
{
    case konstans kifejezés: utasítások; **break**;
    case konstans kifejezés: utasítások; **break**;
    . . .
    default: utasítás; **break**;
}

A **switch** utasítás kiértékeli a *kifejezést* és a megfelelő **case** címkére adja át a vezérlést, amelyben a *konstans kifejezés* értéke megegyezik a *kifejezés* értékével. Amennyiben nem egyezik meg egyik *konstans kifejezéssel* sem, akkor a **default** címkével megjelölt utasítással folytatódik a program futása. Ha nincs **default** címke, akkor a **switch** utasítás blokkját záró } utáni utasításra adódik át.
A *konstans kifejezés* **int** vagy **char** típusú lehet.

Az adott címkéhez tartozó programrészlet végrehajtása után **goto**, **break** vagy **return** utasítással léphetünk ki a **switch** utasításból. Leggyakrabban a **break** utasítást használjuk, mivel ezzel a **switch** utasítás után álló utasítással tudjuk folytatni a programot.

A példában **int** típusú címke szerepel a **switch** utasításban:

A tanulmányi átlag alapján irassuk ki a rendűséget.

```
/* prg_4.cpp */
#include <iostream.h>
void main()
{
 double tanatlag;
 int    jegy;
   cout << "\nA tanulmányi átlag: ";
   cin  >> tanatlag;
   jegy = tanatlag+0.5;
   cout << "A rendűség: ";
   switch(jegy)
   {
     case  1  : cout << "elégtelen"; break;
     case  2  : cout << "elégséges"; break;
     case  3  : cout << "közepes"; break;
     case  4  : cout << "jó"; break;
     case  5  : cout << "jeles"; break;
   }
 }
```

A program futásának eredménye:

```
A tanulmányi átlag: 4.51
A rendűség: jeles
```

1.11.4. Ciklusutasítások

Más nyelvekhez hasonlóan a **for** utasítás szolgál arra, hogy számlálóvezérelt ciklusokat készítsünk. Általános alakja:

> **for**(*inicializáló_kifejezés;*
> *feltétel_kifejezés;*
> *léptető_kifejezés*)
> utasítás vagy blokk

A ciklusba lépve kiértékelődik az *inicializáló_kifejezés*, ezután a *feltétel_kifejezés* értéke meghatározása következik, ha az értéke hamis (0), akkor a ciklus befejezte a munkáját. Egyébként végrehajtásra kerül a ciklus törzsét alkotó utasítás vagy a több utasításból álló blokk; és ezután a *léptető_kifejezés* értékének meghatározására kerül sor majd visszatérünk a ciklusfejben lévő *feltétel_kifejezés* újbóli kiértékelésére, és a ciklus mindaddig folytatódik, amíg a *feltétel_kifejezés* hamis nem lesz.

Számítsuk ki az egész típusú alap egész (pozitív vagy negatív) hatványát! Alkalmazzuk a **for** típusú ciklust!

```
/* prg_5_1.cpp */
#include <iostream.h>
#include <math.h>
void main()
{
 int    i, alap, kitevo;
 double hatvany;
   cout << "alap  : "; cin >> alap;
   cout << "kitevő: "; cin >> kitevo;
   hatvany = 1;
   if( kitevo)
   {
     for(i = 1; i<=abs(kitevo); i++)
       hatvany *= alap;
     if(kitevo < 0)  hatvany = 1./hatvany;
   }
   cout << "Hatvány : " << hatvany;
 }
```

A program futásának eredménye:

```
alap  : 5
kitevő: 2
Hatvány : 25
```

A **for** ciklus fejében lévő kifejezések közül bármelyik el is maradhat. Ha a feltétel_kifejezést hagyjuk el, akkor az állandó igazat jelent. A

```
for(;;)
{
   ...
}
```

alakú ciklus a végtelen ciklus, kilépés csak egyéb vezérlőutasításokkal lehetséges.

Az elöltesztelt, feltételvezérelt ciklusok szervezésére szolgál a **while** utasítás. Alakja:

while (*kifejezés*) utasítás vagy blokk

A fenti ciklusba lépve a *kifejezés* kiértékelésre kerül, és ha értéke 0, akkor a ciklus törzsére nem adódik egyszer sem a vezérlés. Ha a kifejezés igaz, akkor végrehajtásra kerül az utasítás vagy blokk, majd újra a kifejezés kiértékelése következik. A ciklusból tehát mindaddig nem jutunk ki, amíg a ciklusfeltétel hamis nem lesz.

Számítsuk ki az egész típusú alap egész (pozitív vagy negatív) hatványát! Alkalmazzuk a **while** típusú ciklust!

```
/* prg_5_2.cpp */
#include <iostream.h>
#include <math.h>
void main()
{
 int    i, alap, kitevo, n;
 double hatvany;
   cout << "alap  : "; cin >> alap;
   cout << "kitevő: "; cin >> kitevo;
   hatvany = 1;

   if( kitevo )
   {
      n = kitevo;
      if ( kitevo < 0) n = abs(n);
      while (n > 0)
      {
       hatvany *= alap;
       n--;
      }
      if(kitevo < 0)  hatvany = 1./hatvany;
   }
    cout << "Hatvány : " << hatvany;
  }
```

A program futásának eredménye:

```
alap  : 5
kitevő: -2
Hatvány : 0.04
```

Végtelen ciklus hozható létre a

```
     while (1) { .. }
```

utasítással.

Hátultesztelő, feltételvezérlet ciklus az ún. **do-while** ciklus.

> **do**
> utasítás
> **while** (*kifejezés*);

A ciklus törzse egyszer mindenképpen végrehajtódik, amennyiben a kifejezés értéke igaz (nem 0), akkor ciklus újra végrehajtódik, hamis (0) érték esetén a ciklus befejezi a munkáját.

Számítsuk ki az egész típusú alap egész (pozitív vagy negatív) hatványát! Alkalmazzuk a **do..while** típusú ciklust!

```
/* prg_5_3.cpp */
#include <iostream.h>
#include <math.h>
void main()
{
 int    i, alap, kitevo, n;
 double hatvany;
   cout << "alap  : "; cin >> alap;
   cout << "kitevő: "; cin >> kitevo;
   hatvany = 1;
   if( kitevo )
   {
     n = kitevo;
     if ( kitevo < 0) n = abs(n);
     do
     {
       hatvany *= alap;
       n--;
     } while ( n > 0);
     if(kitevo < 0)  hatvany = 1./hatvany;
   }
   cout << "Hatvány : " << hatvany;
 }
```

A program futásának eredménye:

```
alap  : 5
kitevő: 0
Hatvány : 1
```

1.12. Mutatók

A C nyelv központi elemét képezik a mutatók. A pointer aritmetika következtében a C nyelvben a mutatókkal sokféle művelet végezhető. Tulajdonképpen ebben rejlik a C egyik nagy erőssége. A mutatókkal elérhető gyors memóriakezelés és a sokrétű műveletek teszik a C-t assembly nyelv jellegűvé. A mutatók típusos megadása elősegíti a hibamentes programok készítését, de ha szükséges, megfelelő explicit típuskonverzióval kellő rugalmasság biztosítható. A C programok rugalmassága a függvényekre mutató pointerekkel is növelhető. A következőkben lépésről lépésre áttekintjük a mutatókkal kapcsolatos ismereteket.

1.12.1. A mutatók használata

A mutatók a C nyelvben bármilyen típusú adatra, vagy bármilyen típust visszaadó függvényre mutathatnak. A mutató fontos jellemzője a méretén és az értékén kívül az is, hogy pontosan milyen típusra mutat. Természetesen egy összetettebb mutatótípus értelmezésekor könnyen el lehet tévedni.

A pointert változóban deklaráljuk, azonban mivel a pointer tulajdonképpen egy memóriacím és a memóriacím egy szám, mégsem tárolhatunk egy pointert akár egy egész vagy valós típusú változóba. A pointer változó deklarálása hasonlóan történik bármely más változó deklarálásához, kivéve az, hogy a pointert tartalmazó változó nevét a * karakter előzi meg.

Például

```
double *d, a;
int  i, *j;
char c, *ch;
```

amely azt jelenti, hogy a *d* egy pointer változó, amely olyan pointert tud tárolni, amely **double** típusra mutat, az *a* változó duplapontos számot tárol. Az *i* egész változót tárolhat, viszont a *j* olyan pointert tárolhat, amely egész típusra mutat. A *c* karakter tárolására alkalmas, viszont a *ch* pointer karakter típusú változóra mutathat.

1.12.2. A void * típusú mutatók

A C nyelv lehetővé teszi a típus nélküli, ún. általános mutató használatát is:

```
double w;
void *pw = &w;
```

A **void** sohasem jelöl ki memóriaobjektumot. Ha ilyen mutatóval szeretnénk a hivatkozott objektumnak értéket adni, akkor a megfelelő típusra át kell alakítani:

```
*(double*)pv = 2.56;
```

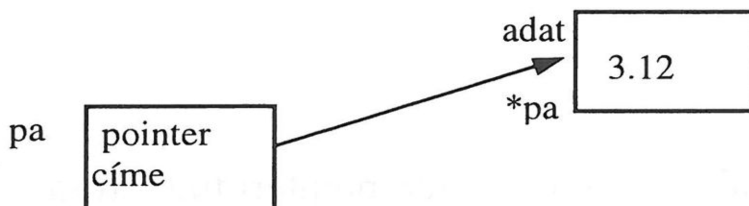
A legtöbb mutatót visszaadó könyvtári függvény a **void*** típussal rendelkezik.

1.12.3. Mutatók értékadása

Egy mutatónak értékül adhatjuk már létező, a mutató típusával azonos típusú objektum címét. Megjegyezzük azonban, hogy általában nem azért használjuk a mutatókat, hogy statikus objektum fix címére mutassunk vele.

```
double *pa, adat;
pa = &adat;
*pa = 3.12;
```

A példában a *pa* pointer változó az *adat* címét veszi át & cím operátor segítségével. A következő utasításban 3.12 értéket írunk arra a memóriaterületre, ahová a *pa* pointer mutat. Ezután az *adat* változó is felveszi a 3.12 értéket, hiszen a *pa* pointer és az *adat* változó címe megegyezik.




```
/* prg_6_1.cpp */
#include <iostream.h>
void main()
{
 double *pa, adat;
   pa = &adat;
   *pa = 3.12;
```

```
    cout << "A pointer létező objektumra mutat\n";
    cout << "adat változó címe                : " << &adat << endl;
    cout << "pa pointer változó tartalma(cím): " << pa << endl;
    cout << "*pa (pointer által mutatott cím tartalma) = " << *pa
         << endl;
    cout << "adat változó tartalma                    = " << adat
         << endl;
}
```

A program futásának eredménye:

```
A pointer létező obektumra mutat
adat változó címe                : 0xffee
pa pointer változó tartalma(cím): 0xffee
*pa (ahová a mutat a pointer) = 3.12
adat változó tartalma         = 3.12
```

A pointerek használatánál a cél a memória dinamikus használata. Ennek során memóriablokkot foglalunk le, melyet a mutató segítségével érhetünk el, és ha nincs szükségünk a lefoglalt területre, felszabadítjuk. A memóriafoglalás és memória felszabadítása könyvtári függvények formájában van jelen a C nyelvben, amelyet a C++ nyelvben is használhatunk.

Dinamikusan memóriaterületet a ***malloc*** függvénnyel foglalhatunk le, melynek a deklarációját az *stdlib.h* állomány tartalmazza.

```
    void *malloc (size_t size)
```

A ***malloc*** függvény a paraméterként megadott mennyiségű bájtot foglal le és visszaadja a lefoglalt területre mutató pointert. Ha nincs elég memória, akkor a visszatérési érték NULL pointer. A ***malloc*** függvénynek paraméterként a kívánt típusnak megfelelő helyet bájtokban mért méretét kell megadni. A különféle típusok méretét a

```
    sizeof(típusnév)
```

segítségével könnyen megkaphatjuk, a típus tárolására szükséges memóriaterületet bájtokban adja vissza.

A ***malloc*** függvény általános mutató típust (**void***) ad vissza, ezt a kívánt pointer típusra át kell alakítani. A *type cast* operátorral olyan típussá alakítjuk át a **void*** pointer típust, amilyen a bal oldal típusa. Ez az átalakítás (kasztolás), amely a függvény előtt kerek zárójelben megadott pointer típusra alakítja át pointert.

A *pa* pointer változó (**double***) azaz duplapontosságú valós változóra mutató pointert tárol, ezért a **void*** pointert **double*** típusúra kell átalakítani:

```
pa = (double*)malloc(sizeof(double));
```

Egy C illetve C++ program memória használata attól lesz dinamikus, hogy a nem használt memóriaterületeket felszabadítja. Ha pointerekre már nincs szükségünk, akkor a lefoglalt területek felszabadításáról a ***free*** függvénnyel gondoskodhatunk, melynek a paramétere a felszabadítandó pointer. A *pa* pointer felszabadítása:

```
free(pa); pa = NULL;
```

Ha a pointer változó tartalma NULL, az azt jelenti , hogy még nincs mögötte tárolásra alkalmas memóriaterület.

```
/* prg_6_2.cpp */
#include <iostream.h>
#include <stdlib.h>
void main()
{
 double *pa, adat;
   adat = 2.4;
   pa = (double*)malloc(sizeof(double));
   *pa = 3.12;
   cout << "pa (cim, amely a lefoglalt memóriaterületre mutat) = "
        << pa << endl;
   cout << "*pa(cím által mutatott memória tartalma)  = " << *pa
        << endl;
   cout << "adat = " << adat << endl;
   free(pa);
}
```

A program futásának eredménye:

```
pa (cim, amely a lefoglalt memóriaterületre mutat) = 0x1470
*pa  = 3.12
adat = 2.4
```

A programban a memóriafoglalási kísérlet (***malloc***) után érdemes megvizsgálni, hogy sikerült-e lefoglalni a kívánt memóriablokkot. Amennyiben kezdőérték nélküli mutató használunk, meg van az esélyünk arra, hogy gépünk "lefagyjon".

```
if (pa == NULL) { cout << "Nincs elég memória!\n"; exit(-1);}
```

Mivel a programnak a hiba esetén -1 lesz a visszatérési értéke (egyébként normál esetben 0-át ad vissza), változtatnunk kell még a programban:

A ***main*** függvény visszatérési típusát **int**-re változtatjuk:

```
int main()
```

és

```
return 0;
```

térünk vissza, amely a jó futási módot jelenti.

```
/* prg_6_3.cpp */
#include <iostream.h>
#include <stdlib.h>

int main()
{
 double *pa, adat;
   adat = 2.4;
   pa = (double*)malloc(sizeof(double));
   if (pa == NULL) { cout << "Nincs elég memória!\n"; exit(-1);}
   *pa = 3.12;
   cout << "pa (cim, amely a lefoglalt memóriaterületre mutat) = "
        << pa << endl;
   cout << "*pa(cím által mutatott memória tartalma)  = " << *pa
        << endl;
   cout << "adat = " << adat << endl;
   free(pa);
   return 0;
}
```

1.13. Tömbök

A C(++) nyelvben az elemi típusokból felépített adatstruktúrákat **aggregátumoknak** nevezzük, és alapvetően háromfélék lehetnek: tömbök, struktúrák és unionok.

A C(++) nyelv megengedi, hogy bármilyen adattípusból tömböt hozhassunk létre. A deklaráció:

azonosító[kifejezés]

más szóval alkalmazzuk a [ ] tömbtípust képző típusmódosító operátort a deklarációnál.

```
int  Xadat[10];
float x[5];
double w[3][5];
char szoveg[20];
char *menu[6];
```

A tömbindexek nullától kezdődnek, így az egész értékeket tartalmazó *Xadat* tömbnek 0 és 9 között van érvényes indexe, a 10-edik elemre való hivatkozás hibát okoz. A példában látható, hogy a **float**, **double** és a **char** típusokon kívül a *menu* 6 elemű karaktermutató tömböt deklarál.

A többdimenziós tömböknek a dimenzióit külön [ ] zárójelbe kell tenni. Kétdimenziós mátrix esetében

```
double w[3][5];
```

A *w* három sorból és négy oszlopból áll.

Kezdőértéket az alábbi módon adhatunk

```
int tomb[][2] = {
                  { 1, 3 },
                  { 4, 2 },
                  { 6, 8 }
                };
```

Az egész típusú *tomb* változó a definíció hatására egy 3 sorból és 2 oszlopból álló kétdimenziós tömb lesz a fenti kezdőértékekkel. Figyeljük meg, hogy az indexméretek közül csak az elsőt hagytuk el, a fordítóra bízva annak meghatározását a kezdőértékek alapján. A második (és esetleges további) indexhatárok megadása mindig kötelező.

```
/* prg_7_1.cpp */
#include <iostream.h>
void main()
{
 int tomb[][3] = {
                   { 1, 3 },
                   { 4, 2 },
                   { 6, 8 }
                 };
 int i,j;
```

```
    cout << "A tömb elemei\n";
    for(i = 0; i<3; i++)
    {
      for(j = 0; j<2; j++)
        cout << tomb[i][j] << " ";

    }
}
```

A program futásának eredménye:

```
A tömb elemei
1 3 4 2 6 8
```

1.13.1. Kapcsolat tömbök és mutatók között

A pointerek és a tömbök közötti szoros kapcsolatot az alábbi példák mutatják be:

```
        double vekt[20], *pv = &vekt[0];
```

Az aritmetikai szabályok figyelembevételével ezek után a következő párosításokat írhatjuk fel:

```
*(pv+0)          vekt[0]
*(pv+1)          vekt[1]
*(pv+2)          vekt[2]
```

ahol tehát az egyes párosok mindkét tagja ugyanarra a tárolási egységre hivatkozik. Ha most egy pillanatra elfelejtjük, hogy *pv* mutató, akkor a fenti analógia a következőképpen tehető teljessé:

```
pv[0]            vekt[0]
pv[1]            vekt[1]
pv[2]            vekt[2]
```

Ez összhangban van azzal, hogy a tömbbeli indexek az elemeket éppúgy sorszámozzák meg, mint ahogy a pointer aritmetikánál a memória felosztását elképzeltük, azaz egy előrelépés az indexben egy adattal való előrehaladást eredményez a memóriában. A *vekt* tömb tehát a *pv* mellé elképzelt memóriastruktúrában helyezkedik el, azaz a kettőt mintegy egymásra fektettük. A fenti analógia érdekében a C(++) nyelv megengedi azt, hogy a pointereket indexelhessük a fent leírt módon, precízen megfogalmazva, a *pointer[egész]* alakú kifejezést a fordító **(pointer + egész)* értelemben alkalmazza. Azért, hogy az indexkifejezés értelmezése független legyen attól, hogy tömbre vagy mutatóra alkalmazzuk, a *vekt[0]* értelmezése is legyen **(vekt+0),* azaz **vekt*,

tehát *vekt* önmagában a *vekt[]* tömb első elemére mutató pointer: *vekt=&vekt[0]*. A kifejezésben bárhol előforduló tömbazonosítót a fordító azonnal átalakítja a tömb első elemét megcímző mutatóvá, és ha indexkifejezés követi, akkor azt a pointerek indexelésének szabályai szerint értelmezi.(Innen is látszik, hogy ilyen összefüggésben a [ ] tényleg az indexelő operátor).

A mutatók és a tömbök közötti szoros kapcsolat teszi lehetővé, hogy egydimenziós indexhatárral kezelhessük, azaz a méretüket elegendő futási időben rögzíteni. Például a fenti *vekt* használható a következő formában:

```
double *vekt, *seged;
seged = vekt;
```

ahol a **seged* megfelel a vekt*[0]* elemmel és így tovább...

```
/* prg_7_2.cpp */
#include <iostream.h>
#include <stdlib.h>
void main()
{
 double *vekt, *seged, szum1 = 0, szum2 = 0;
 int i,n;
   cout << "Az adatok száma: "; cin >> n;
   vekt = (double*)malloc(n*sizeof(double));
   for(i = 0; i<n; i++)
   {
     cout << i << " adat: "; cin >> vekt[i];
     szum1 +=vekt[i];
   }
   cout << "összeg: " << szum1 << endl;
   cout << "\núj adatok: " << endl;
   for(i = 0, seged = vekt; i<n; i++, seged++)
   {
     cout << i << " adat: "; cin >> *seged;
     szum2 += *seged;
   }
   cout << "összeg: " << szum2;
   free(vekt);
 }
```

A program futásának eredménye:

```
Az adatok száma: 3
0 adat: 1
1 adat: 2
2 adat: 3
összeg: 6
```

```
új adatok:
0 adat: 4
1 adat: 5
2 adat: 6
összeg: 15
```

A kétdimenziós tömböket a ** pointer segítségével tehetjük dinamikussá. Először helyet foglalunk a sorok számának, majd minden sorelemben újra helyet foglalunk az oszlopok számára.

```
/* prg_7_3.cpp */
#include <iostream.h>
#include <stdlib.h>
void main()
{
  int **mv, i, j, n = 2, m = 3;
   mv = (int**)malloc(n*sizeof(int*));
   for( i = 0; i<n; i++)
   mv[i] = (int*)malloc(m*sizeof(int));
   for( i = 0; i<n; i++)
   for( j = 0; j<m; j++)
   {
     cout << i << "," << j << ".adat :";
     cin >> mv[i][j];
   }
   cout << "A beolvasott adatok: " << endl;
   for( i = 0; i<n; i++)
   {
     for( j = 0; j<m; j++)
       cout << "[" << i << "," << j << "] = " << mv[i][j] << "    ";
     cout << endl;
   }
   for( i = 0; i<n; i++)
     free(mv[i]);
   free(mv);
  }
```

A program futásának eredménye:

```
0,0.adat : 1
0,1.adat : 2
0,2.adat : 3
1,0.adat : 4
1,1.adat : 5
1,2.adat : 6
A beolvasott adatok:
[0,0] = 1    [0,1] = 2    [0,2] = 3
[1,0] = 4    [1,1] = 5    [1,2] = 6
```

1.14. Struktúrák és unionok

Az eddig megismert egyetlen aggregátum, az egydimenziós tömb, a következő előnyöket biztosítja: tetszőleges eleméhez azonos idő alatt lehet hozzáférni, mérete dinamikusan is beállítható, sőt tetszőleges folytonos része önálló tömbként is kezelhető. Hátránya, hogy minden elemének azonos típusúnak kell lennie. A most ismertetésre kerülő aggregátum, a struktúra viszont lehetőséget biztosít arra, hogy logikailag összetartozó, de különböző típusú változókat egységbe fogva kezelhessünk. Minden definiált adatstruktúra a fordító számára az új típusként jelentkezik (**typedef** -fel név is rendelhető egy struktúra típushoz), és a továbbiakban azonos módon használhatjuk, mint az elemi típusokat (azaz deklarálhatunk illetve definiálhatunk ilyen típusú objektumokat, alkalmazhatjuk rá a **sizeof** operátort, részt vehet típusmódosító szerkezetben, képezhetünk ilyen típusra mutató pointereket is stb.). A korszerű C implementációk – mint például a Borland C++ – lehetővé teszik (azonos típusú) struktúraváltozók közt a közvetlen értékadást, struktúrák szerepeltetését függvények paramétereként illetve visszatérési értékként is.

1.14.1. Struktúrák megadása

A struktúra (**struct**) típus több, tetszőleges (kivéve a **void** és a függvénytípus) objektum együttese. Először deklarálnunk kell a struktúra típust, melyet felhasználva változókat deklarálhatunk. A struktúra deklarációjának általános formája:

tárolási osztály **struct** *típuscímke* {
típus1 azonosítólista1;
típus2 azonosítólista2;
. . .
}*azonosítólista*;

Az **enum** deklarációhoz hasonlóan elmaradhat a típuscímke (a struktúra megnevezése), ha a későbbiekben nem kívánunk hivatkozni rá, illetve elmaradhat az *azonosítólista* is, ha csak a struktúra alakját kívánjuk megadni. A struktúrát felépítő elemek, a mezők (*members*) deklarálása típusuk és nevük megadásával történik. A típuscímke tipikus alkalmazása az önhivatkozó (rekurzív) adatstruktúrák definiálása.

```
struct adatok {
                int kor;
                float jutalom;
                char nev[30];
                struct adatok *kovetkezo;
              } sa, *ps;
```

Struktúra változót az alábbiakban is létrehozhatunk:

```
struct típuscimke  struktúra_változó;
```

A típusdefiníciót használva:

typedef struct *típuscímke* {
típus1 azonosítólista1;
típus2 azonosítólista2;
. . .
} *TIPUS*;

A deklaráció:

```
TIPUS  stváltozó1, stváltozó2;
```

1.14.2. Hivatkozás a struktúra elemekre

Ha egy struktúra elemére kívánunk hivatkozni, akkor a . (pont) mezőkiválasztó operátort használhatjuk. Például

```
sa.kor= 35;   (*ps).jutalom=1000;
```

Mivel pointerekkel gyakran mutatunk struktúrákra, a fenti utolsó példának megfelelő hivatkozási forma sűrűn előfordul. Erre az esetre egy új operátort bocsátottak rendelkezésünkre a nyelv tervezői, ez a -> operátor (mínusz jel és nagyobb jel). A bal oldalon struktúra mutató pointernek, a jobb oldalon pedig az adott struktúra egy mezőazonosítójának kell állnia. A

kifejezés->azonosító

forma teljesen megegyezik az alábbi alakkal:

*(*kifejezés).azonosító*

Tehát a fenti utolsó példánkat a következőképpen is írhattuk volna:

```
ps-> jutalom = 1000;
```

A tanuló nevére és átlagára hozzunk létre struktúrát, majd a struktúra definiálására hozzunk létre egy TANULOTIP saját típust.

```
/* prg_8_1.cpp */
#include <iostream.h>
typedef struct tanulo{
                        char   neve[20];
                        double atlaga;
                       } TANULOTIP;

void main()
{
 TANULOTIP t;
   cout << "Neve  : "; cin >> t.neve;
   cout << "átlaga: "; cin >> t.atlaga;
   cout << "A tanuló adatai\n";
   cout << t.neve << "  " << t.atlaga << endl;
}
```

A program futásának eredménye:

```
Neve  : Laszlo
átlaga: 4.5
A tanuló adatai
Laszlo  4.5
```

Definiáljunk a meglévő TANULOTIP típusból *tomb* típust, amely alkalmas 10 tanuló adatainak a tárolására.

```
/* prg_8_2.cpp */
#include <iostream.h>
typedef struct tanulo{
                        char   neve[20];
                        double atlaga;
                      } TANULOTIP;
typedef TANULOTIP tomb[10];
void main()
{
 tomb t;
 int  i, n;
   cout << "Tanulók száma: ";
   cin >> n;
```

```
  for( i = 0; i<n; i++)
  {
    cout << "Neve   : "; cin >> t[i].neve;
    cout << "átlaga: "; cin >> t[i].atlaga;
  }
  cout << "A tanulók adatai\n";
  for(i = 0; i<n; i++)
  {
     cout << t[i].neve << "  " << t[i].atlaga << endl;
  }
 }
```

A program futásának eredménye:

```
Tanulók száma: 3
Neve : Laszlo
átlaga: 4.5
Neve   : Szilvi
átlaga: 4.2
Neve   : Zoltan
átlaga: 4.7
A tanulók adatai
Laszlo  4.5
Szilvi  4.2
Zoltan  4.7
```

Tervezzünk műveletvégzésre alkalmas struktúrát, amely két valós adatot, a műveleti jelet és az eredményt tárolja.

```
/* prg_8_3.cpp */
#include <iostream.h>
typedef struct Muveletvegzes{
                              float x,y;
                              float eredmeny;
                              char  muv;
                             } MUVELET;
void main()
{
 MUVELET z;
  cout << "művelet (+,-,*,/): "; cin >> z.muv;
  cout << "1. adat: "; cin >> z.x;
  cout << "2. adat: "; cin >> z.y;
  z.eredmeny = 0;
  switch (z.muv)
  {
   case '+' : z.eredmeny = z.x+z.y; break;
   case '-' : z.eredmeny = z.x-z.y; break;
   case '*' : z.eredmeny = z.x*z.y; break;
   case '/' : z.eredmeny = z.x/z.y; break;
  }
```

```
  cout << z.x << " " << z.muv << " " << z.y
       << " =  " << z.eredmeny;
 }
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 2.5
2. adat: 3.1
2.5 * 3.1 =  7.75
```

Tervezzünk műveletvégzésre alkalmas struktúrát, amely két valós adatot, a műveleti jelet és az eredményt tárolja. A feladatot a struktúrára mutató pointerrel oldjuk meg.

```
/* prg_8_4.cpp */
#include <iostream.h>
#include <stdlib.h>
typedef struct Muveletvegzes{
                                float x,y;
                                float eredmeny;
                                char  muv;
                              } MUVELET;
void main()
{

 MUVELET *z;
   z = (MUVELET*) malloc(sizeof(MUVELET));
   cout << "művelet (+,-,*,/): "; cin >> z->muv;
   cout << "1. adat: "; cin >> z->x;
   cout << "2. adat: "; cin >> z->y;
   z->eredmeny = 0;
   switch (z->muv)
   {
         case '+' : z->eredmeny = z->x+z->y; break;
         case '-' : z->eredmeny = z->x-z->y; break;
         case '*' : z->eredmeny = z->x*z->y; break;
         case '/' : z->eredmeny = z->x/z->y; break;
   }
        cout << z->x << " " << z->muv << " " << z->y
         << " =  " << z->eredmeny;
  free (z);
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 2.5
2. adat: 4
2.5 * 4 =  10
```

Hozzunk létre egy *Aru* struktúrát, amely tartalmazza az áru azonosítóját, az árát és a darabszámát. Az adatok beolvasásához és kiírásához definiáljuk át az *ostream* és az *istream* operátorokat.

```
/* io8.cpp */
#include <iostream.h>
typedef struct Aru {
                         char *aru_azonosito;
                         float ara;
                         int db;
                       };
ostream& operator << (ostream& t, Aru& a)
{
  t << a.aru_azonosito << " " << a.ara << " " << a.db;
  return t;
}
istream& operator >> (istream& t, Aru& a)
{
  t >> a.aru_azonosito >> a.ara >> a.db;
  return t;
};

void main()
{
 Aru r;
   r.aru_azonosito = new char[20];
   cout << "\nÁru, ára, db: ";
   cin >> r;
   cout << r << "\n";
   cout << "Áru: " << r;
}
```

A program futásának eredménye:

```
Áru, ára, db: asztal 10 1520
Áru: asztal 10 1520
```

1.14.3. A bitmezők

A struktúrák másik felhasználási területe, hogy segítségükkel felbonthatunk sorszámozott típusú adatokat **char**-nál is rövidebb bithosszúságú részekre. Ilymódon még arra is van lehetőségük, hogy egy **int** értéknek minden egyes bitjét külön-külön kezelhessük. Az ilyen, bithosszban meghatározott elemek neve bitmező (*bit field*) és hosszukra az egyetlen megkötés, hogy minden bitmezőnek el kell férni egy **int** tárolására szolgáló területen. Használatuknak akkor van jelentősége, ha nagy mennyiségű jelzőt (*flag*-et) szeretnénk minél tömörebben tárolni, vagy ha hardverközeli programozásban az egyes biteknek,

bitcsoportoknak különálló funkciója van, és egymástó függetlenül kívánjuk ezeket használni. Jóllehet, ez a lehetőség eddig is rendelkezésünkre állt, hiszen a bitenkénti operátorokkal tetszőleges bitet kiválaszthattunk, beállíthattunk illetve törölhettünk, de a bitmezők használata ugyanerre kényelmesebb és jobban követhető lehetőséget biztosít. A bitmezőket formailag struktúraelemként kell definiálni és használni, az egyetlen eltérés az, hogy a bitmezők azonosítóját kettősponttal (:) elválasztva a bitben kifejezett hossz megadása követi:

típusazonosító <bitmező azonosító> : hossz;

ahol a bitmezők típusa **char**, **usigned char**, **int** vagy **unsigned int** lehet. A bitmezők a szónak illetve bájtnak alacsonyabbtól a magasabb helyiérték bitmezőit foglalják le.

tárolási osztály spec. **struct** *típuscimke*
{
típus_1 azonosító 1: hossz_1;
típus_1 azonosító 1: hossz_1;
. . .
}*azonosítólista;*

Eredetileg a bitmezőket előjel nélkülinek tervezték, függetlenül a megadott típustól; a Borland C++ azonban az **int**-ként definiált bitmezőket előjeles mennyiségként kezeli. Mindaddig, amíg az egymás után következő bitmezők elférnek egy gépi szóban, a fordító ezeket egy (**unsigned** vagy **signed**) **int**-be kapja, egyébként újat kezd. Lehetnek meg nem nevezett bitmezők is, csak kettőspontból (:) és hosszból felépítve, a nem használt helyek kitöltésére. A 0 speciális hosszmegadás befejezi az adott gépi szó bitmezőkkel való feltöltését, és újat kezd.
Például:

```
struct bitmezo {
                  int         k: 2;
                  unsigned    i: 4;
                  int          : 5;
                  int         n: 1;
                  unsigned    m:4;
                } f1,f2;
```

A bitek kiosztása az alábbi:

1.16. táblázat

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
x	x	x	x	x	x	x	x	x	x	x	x	x	x	x	x
←			→	↔	←				→	←			→	←	→
m				n	használatlan					i				k	

1.14.4. A union fogalma

Egyes feladatok megoldása során felmerülhet annak igénye, hogy ugyanezt a tárterületet különböző időpontokban különböző típussal vagy jelleggel értelmezzük és használjuk. Vegyük például egy bináris fa felépítésénél használható adatstruktúrát:

```
struct fa{
            unsigned jelzo_1 : 2,
                     jelzo_2 : 2,
                     jelzo_3 : 2;
            long info;
            struct fa *jobb,
                      *bal;
           };
```

Minden csomópontnak tartalmaznia kell a leszármazottra mutató pointereket. A leveleket (végcsomópontokat) arról lehet felismerni, hogy mindkét leszármazottuk hiányzik, tehát a jobb- és a baloldali mutatók speciális "nem használt" jelzést adó értékkel vannak feltöltve (ami, mint látni fogjuk majd, a NULL érték). A levelekre általában jellemző, hogy a többi csomóponthoz képest további információt hordoznak. Ilymódon a fenti adatstruktúra a levelek számára nem megfelelő, mert ezt a többletadatot nem képes hordozni. Viszont, ha egy fa új elemmel bővül, akkor valamelyik korábbi levél elágaztató csomóponttá válhat. Nehézkes lenne ehhez megváltoztatni típusát és ezzel együtt méretét. Most vehetjük hasznát annak, hogy a leveleknek nincs utódjuk, tehát a bal és jobb pointer együtt – 32 vagy 64 biten – hordozza azt az egybites információt, hogy ez egy levél. Ha felveszünk a meglévő bitmezők közé egy újabbat, akkor felszabadíthatnánk a két mutató helyét a pótlólagos információ számára, csak a fordítót kell értesíteni róla, hogy ilyenkor ezeket más módon kívánjuk értelmezni. Erre szolgál az **union**.

Példánk átírva:

```
struct fa{
        unsigned jelzo_1 : 2,
                 jelzo_2 : 2,
                 jelzo_3 : 2;
                 level   : 1;
        long info;
        union {
                struct {
                        struct fa *jobb,
                                  *bal;
                       }utodok;
                long levelinfo;
              } u;
    };
```

A fenti felírás azt jelenti a fordítóprogram számára, hogy az *u*-val jelölt adatterület kétféleképpen is használható: vagy két pointert kell ezen a helyen tárolni az *utodok* megnevezés alatt, vagy egy hosszú egész értéket *levelinfo* néven. Ezért a fordító az *u* típusú adatok tárolásához akkora helyet választ, amekkora garantáltan elegendő ahhoz, hogy mindkét résztípust külön-külön (de nem egyszerre) be tudja fogadni.
Ha a *cap* típusa a fenti struktúra, akkor a levél esetén írható a következő értékadás:

```
cap.u.levelinfo = k;
```

míg ha a levélből csomópont lett

```
cap.u.utodok.bal = utod;
```

feltéve, hogy *k* hosszú egész, *utod* pedig a fenti struktúrájú adatra mutató pointer. A **union**-ok definíciója és használata formailag teljesen megegyezik a struktúrákkal, csak azt kell figyelembe venni, hogy a struktúrákat a felsorolt elemek egyszerre, együttesen alkotják, míg a **union**-okban egyidőben a felsorolt elemek közül csak egy számára van hely. Fontos, hogy mindig az adat típusának megfelelő módon használjuk.

1.15. Függvények

Az eddigiekben tárterület foglaló tárolási egységek definiálásáról és deklarációjáról volt szó, most tekintsük át a kódgeneráló program egységeket ebből a szempontból. A függvény definíciójának általános formája:

tárolási_osztály típus azonosító (formális paraméterlista)
{
<lokális definíciók és deklarációk>
utasítások
return (*visszatérési érték*);
}

blokk:
a függvény törzse

- A *tárolási_osztály* specifikátor vagy üres (globális függvény), vagy **static** (modulra lokális függvény).
- A *típus* a függvény által visszaadott érték típusa, ez gyakorlatilag tetszőleges lehet (kivétel, hogy függény nem adhat vissza tömböt vagy függvényt, de visszaadhat ezeket megcímző pointert).
- A függvényazonosítóra (*azonosító*) az azonosítókról általában elmondottak érvényesek.
- A ( ) zárójelek között álló *formális paraméterlista* a paraméterek deklarációjára szolgál, ahány deklaráció szerepel benne, annyi paramétere van a függvénynek (akár egy sem).
- A *blokk* – {} zárójelek között – deklarációkat, blokkra lokális adatdefiníciókat és végrehajtható utasításokat tartalmazhat.

A **return** utasítás feladatai:

- Befejezi az éppen futó függvény működését és a vezérlés visszakerül a hívó függvényhez.
- **return**; alakban használva olyan függvényből léphetünk ki, amely a nevével nem ad vissza semmilyen értéket (**void**).
- A függvények többsége valamilyen értékkel tér vissza a hívás helyére, melynek értékét a return utasításban definiálhatjuk:
  return *kifejezés;*.

A függvények fajtái:

1. A függvény a paraméterlistáján átvett adatok értékeivel számol és egyetlen eredményt ad vissza a **return** segítségével. A függvény aktiválása történhet az értékadó utasítás jobb oldalán, ***printf*** függvényben, vagy ***cout*** output utasításban kiíratható a visszaadott érték.

   Az *fgv1* függvénynek két egész típusú bemenő paramétere van (*x,y*) és egész típusú eredményt szoltáltat:

```
int fgv1(int x, int y);
. . .
cout << "eredmény: " << fgv1(3,4);
```

2. A függvénynek nincs (**void**) visszatérési értéke, de a paraméterlistáján pointer változó megadásával, (vagy referencia által) a függvény több értéket is tud vissza tud adni. Ezeknek a függvényeknek az aktiválása hasonlít más nyelvekben lévő eljárások hívásához. Mivel a függvénynek nincs visszatérési értéke, ezért nem aktiválható értékadás jobb oldalán.
   Az *fgv2* függvénynek két egész típusú bemenő paramétere van (*a,b*) és az eredményt a *c* egész típusra mutató pointer által adja vissza:

```
void fgv2(int a, int b, int *c);
```

3. A függvény a paraméterlistáján is ad vissza értékeket és visszatérési értékkel is rendelkezik.
   Az *fgv3* függvénynek egy duplapontosságú (*w*) és egy egész típusú (*n*) bemenő paramétere van. A függvény két ereményt ad vissza: a *z* duplapontos típusú pointerrel az egyiket és a **return** mellett visszatérési értékként a másikat

```
double fgv3(double w, int n, double *z);
```

4. A függvénynek nincsenek paraméterei és nincs visszatérési értéke sem. A paraméterlistán nem kötelező a **void** típust kitenni.

```
void fgv3(void);
```

   vagy

```
void fgv4();
```

5. Ha a formális paraméterlistában **...** áll az utolsó paraméter helyett, akkor ezzel azt jelezzük, hogy az adott függvénynek a deklaráltakon kívül további, ismeretlen számú és/vagy ismeretlen típusú paramétere lehet.

Erre jellemző példa az *stdio.h* fejléc fájlban deklarált

```
int printf(char *,... );
```

standard függvény (mely az első paraméterében adott sztringben lévő specifikáció szerint a szabványos kimenet állomány – tipikusan a képernyőre – nyomtatja ki a további paramétereit.)

1.15.1. Előredefiniált függvények használata

A példaprogramban két szám mértani közepének kiszámításához felhasználjuk az ***sqrt*** előredefiniált függvényt. A függvény deklarációja a *math.h* állományban van, melyet a programba kell beszerkesztenünk a *#include* előfordító utasítással, különben hibajelzést kapunk:

Function 'sqrt' should have a prototype

A függvénynek legalább egy paraméterrel kell rendelkeznie, vagy van olyan előredefiniált függvény, amely egynél több paraméterrel működik. Például számítsuk ki 3.5 -nek 4-dik hatványát a ***pow*** függvénnyel, melynek az első paramétere az alap, a második paramétere a kitevő, visszatérési értéke a kívánt hatvány:

```
cout << "3.5-nek 4-dik hatványa: " << pow(3.5,4.0);
```

A paraméter lehet konstans, változó és kifejezés is.

```
/* prg_9_1.cpp */
#include <iostream.h>
#include <math.h>
void main()
{
 double a1 = 6.0 , a2 = 4.5, szorzat, ered1, ered2;
  szorzat = a1 * a2;
  ered1 = sqrt(szorzat);
  ered2 = sqrt(a1*a2);
  cout << "Mértani közép: " << ered1 << endl;
  cout << "Mértani közép: " << ered2 << endl;
  cout << "Mértani közép: " << sqrt(a1*a2) << endl;
  cout << "Mértani közép: " << sqrt(27) << endl;
}
```

A program futásának eredménye:

```
Mértani közép: 5.196152
Mértani közép: 5.196152
Mértani közép: 5.196152
Mértani közép: 5.196152
```

1.15.2. Típusváltoztató (type cast) operátor

Ha két egész számot osztunk: 3/2 , eredmény 1 és nem 1.5 lesz. Ha valós eredményt szeretnénk kapni, akkor valamelyik számnak valósnak kell lennie: 3.0/2 vagy 3/2.0, ezt számkonstans esetén a nulla tized megadásával megoldhatjuk. Ha egész típusú változókkal történik az osztás, akkor az osztandót vagy az osztót kell átalakítanunk.

A C++ nyelvben könnyen konvertálhatunk **int** típusból **double** típusra

```
double(3)/2
3/double(2)
```

Amint látható a **double** típusazonosító, mintha egy előredefiniált függvény lenne, úgy használható az őt követő kifejezés típusának átalakítására. Ez az ún. *type casting,* azaz típusátalakítás vagy másnéven típuskonverzió. A típuskonverzió általános formája:

új_típus (kifejezés)

amely kifejezés típusát az új_típus-sá alakítja.
Példaként értékeljük ki az alábbi mintaprogramot.

```
/* prg_9_2.cpp */
#include <iostream.h>
void main()
{
 int a1 = 3, b1 = 2;
 double ered1, ered2, ered3;
   ered1 = a1/b1;
   cout << "ered1 = " << ered1 << endl;
   ered2 = double (a1)/b1;
   cout << "ered2 = " << ered2 << endl;
   ered3 = a1/ double(b1);
   cout << "ered3 = " << ered3;
 }
```

A program futásának eredménye:

```
ered1 = 1
ered2 = 1.5
ered3 = 1.5
```

1.15.3. Programozó által definiált függvények

Írjunk *MertaniKozep* függvényt két változó mértani közepének kiszámítására. A függvény paraméterként kapja a két változót és eredményül a mértani közepet adja vissza. Írjunk még egy *EredmenyKiir* függvényt, amely a paraméterként kapott változó tartalmát a Mértani közép szöveg mellett írja ki

azonosító — paraméterek

```
double MertaniKozep( double w1, double w2)
{
return (sqrt(w1*w2));
}
```

a függvény törzse

a visszatérési érték

A **void** *EredmenyKiir* függvény nem ad vissza értéket, csak a paraméterlistán átvett változó tartalmát írja ki.

```
/* prg_9_3.cpp */
#include <iostream.h>
#include <math.h>
double MertaniKozep( double w1, double w2)
{
   return (sqrt(w1*w2));
}
void EredmenyKiir( double eredmeny)
{
   cout << "Mértani közép: " << eredmeny << endl;
}
void main()
{
 double a1, a2, ered;
   cout << "1. adat: "; cin >> a1;
   cout << "2. adat: "; cin >> a2;
   ered = MertaniKozep( a1,a2); // a függvény aktiválása
   EredmenyKiir(ered);   // a függvény aktiválása
}
```

A program futásának eredménye:

```
1. adat: 3
2. adat: 4
Mértani közép: 4.582576
```

A *MertaniKozep* és az *EredmenyKiir* függvények a **main** függvény előtt írtuk meg, ezáltal a függvények megelőzték a hívási helyüket, ez egyben a függvények deklarációja is.

Módosítsuk a PRG_9_3.CPP programot, hogy a függvényeket helyezzük a **main** függvény után. Ebben az esetben a függvények definíciói később vannak, mint a hívás helye, ezért a függvényeket deklarálni kell, különben hibajelzést kapunk. A függvények ún. prototípusát kell a **main** függvény előtt felsorolni:

```
double MertaniKozep( double w1, double w2);
void EredmenyKiir( double );
```

A prototípus tulajdonképpen a függvény feje pontosvesszővel lezárva. Nem szükséges a paraméterek neveit megadni, elég csak a típusait vesszővel elválasztva felsorolni:

```
double MertaniKozep( double, double);
void EredmenyKiir( double );
```

```
/* prg_9_4.cpp */
#include <iostream.h>
#include <math.h>
double MertaniKozep( double w1, double w2);
void EredmenyKiir( double );
void main()
{
  double a1, a2, ered;
   cout << "1. adat: "; cin >> a1;
   cout << "2. adat: "; cin >> a2;
   ered = MertaniKozep( a1,a2);
   EredmenyKiir(ered);
}
double MertaniKozep( double w1, double w2)
{
  return (sqrt(w1*w2));
}
void EredmenyKiir( double eredmeny)
{
  cout << "Mértani közép: " << eredmeny << endl;
}
```

A program futásának eredménye:

```
1. adat: 3
2. adat: 4
Mértani közép: 3.464102
```

Írjunk olyan *KozepSzamitas* függvényt, amely vagy mértani vagy számtani középértéket számít ki.

A két valós változó paraméter mellett bevezetünk egy **char** típusú paramétert, amelynek két fajta értéke lehet:

m esetén mértani közép számítása,
s esetén pedig számtani közép számítása.

```
/* prg_9_5.cpp */
#include <iostream.h>
#include <math.h>
double KozepSzamitas( char flag, double w1, double w2);
void EredmenyKiir( double );
void main()
{
 double a1, a2, ered;
 char tipus;
   cout << "1. adat: "; cin >> a1;
   cout << "2. adat: "; cin >> a2;
   do
   {
     cout << "Középszámítás: mértani (m), számtani (s): ";
     cin >> tipus;
   }while (tipus != 'm' && tipus != 's');
   ered = KozepSzamitas(tipus,a1,a2);
   EredmenyKiir(ered);
}
double KozepSzamitas( char flag, double w1, double w2)
{
 double q;
  switch (flag)
  {
    case 'm': q = sqrt(w1*w2); break;
    case 's': q = (w1 + w2)/2; break;
  }
  return (q);
}
void EredmenyKiir( double eredmeny)
{
  cout << "Mértani közép: " << eredmeny << endl;
}
```

A program futásának eredménye:

```
1. adat: 3
2. adat: 4 Középszámítás: mértani (m), számtani (s): m
Mértani közép: 3.464102
```

Bővítsük a *KozepSzamitas* függvény paraméterlistáját a középérték eredményének visszaadásával.

A *KozepSzamitas* függvény paraméterlistája pointer változóval bővül:

```
void KozepSzamitas( char flag, double w1, double w2,
double *kozepertek);
```

A *kozepertek* változó **double** típusú változóra mutató pointert képes tárolni, ebben a változóban tudjuk visszaadni az eredményt.
Ha a főprogramban deklarált változó nem pointer, akkor a függvény hívásakor annak a változónak a címét kell átadni:

```
double a1,a2,ered;
      . . .
KozepSzamitas(tipus,a1,a2,&ered);
```



```
/* prg_9_6.cpp */
#include <iostream.h>
#include <math.h>
void KozepSzamitas( char flag, double w1, double w2,
                    double *kozepertek);
void EredmenyKiir( double );
void main()
{
 double a1, a2, ered;
 char tipus;
   cout << "1. adat: "; cin >> a1;
   cout << "2. adat: "; cin >> a2;
   do
   {
     cout << "Középérték számítása: mértani (m), számtani (s): ";
     cin >> tipus;
   }while (tipus != 'm' && tipus != 's');
   KozepSzamitas(tipus,a1,a2,&ered);
   EredmenyKiir(ered);
}
void KozepSzamitas( char flag, double w1, double w2,
double *kozepertek)
{
   switch (flag)
  {
    case 'm': *kozepertek = sqrt(w1*w2); break;
    case 's': *kozepertek = (w1 + w2)/2; break;
  }
}
void EredmenyKiir( double eredmeny)
{
   cout << "Mértani közép: " << eredmeny << endl;
}
```

A program futásának eredménye:

```
1. adat: 3
2. adat: 4 Középérték számítása: mértani (m), számtani (s): s
Mértani közép: 3.5
```

Módosítsuk a főprogramot úgy, hogy az eredményt tároló változó **double*** típusú pointer legyen.

Ha a programban **double** * pointert deklarálunk a függvény aktiválására, akkor a függvény hívása előtt memóriahelyet kell a pointer változó számára foglalni. A ***malloc*** memóriahelyfoglaló prototípusa az *stdlib.h* állományban van deklarálva:

```
#include <stdlib.h>
```

Az *ered* változó pointer típusú, deklarációja:

```
double a1, a2, *ered;
```

Az *ered* változó számára memóriahely foglalása:

```
ered = (double*)malloc(sizeof(double));
```

Módosul a *KozepSzamitas* függvény hívása:

```
KozepSzamitas(tipus,a1,a2,ered);
```

Módosul az *EredmenyKiir* függvény hívása:

```
EredmenyKiir(*ered);
```

Az *ered* pointert fel kell szabadítani:

```
free(ered);
```

```
/* prg_9_7.cpp */
#include <iostream.h>
#include <math.h>
#include <stdlib.h>
void KozepSzamitas( char flag, double w1, double w2,
double *kozepertek);
void EredmenyKiir( double );
```

```
void main()
{
 double a1, a2, *ered;
 char tipus;
   ered = (double*)malloc(sizeof(double));

   cout << "1. adat: "; cin >> a1;
   cout << "2. adat: "; cin >> a2;
   do
   {
     cout << "Középérték számítása: mértani (m), számtani (s): ";
     cin >> tipus;
   }while (tipus != 'm' && tipus != 's');
   KozepSzamitas(tipus,a1,a2,ered);
   EredmenyKiir(*ered);
   free(ered);
}
void KozepSzamitas( char flag, double w1, double w2,
double *kozepertek)
{
  switch (flag)
  {
    case 'm': *kozepertek = sqrt(w1*w2); break;
    case 's': *kozepertek = (w1 + w2)/2; break;
  }
}
void EredmenyKiir( double eredmeny)
{
  cout << "Mértani közép: " << eredmeny << endl;
}
```

A program futásának eredménye:

```
1. adat: 4
2. adat: 5 Középérték számítása: mértani (m), számtani (s): s
Mértani közép: 4.5
```

Duplapontosságú 20 elem tárolására alkalmas tömb részére definiáljunk saját típust *vekt* néven. Írjunk *Olvas* függvényt, amely adott számú adattal feltölti a tömböt, *Osszegszamitas* függvényt, amelyek kiszámítja a tömb adatainak öszszegét és *Atlagszamitas* függvényt, amely az átlagot számítja ki.

```
/* tomb_1.cpp */
#include <iostream.h>
typedef double vekt[20];
void Olvas(vekt x, int *db);
double Osszegszamitas( vekt x, int db);
double Atlagszamitas( vekt x, int db);
```

```
void main()
{
 vekt s;
 int n;
 double osszeg, atlag;
   Olvas(s,&n);
   osszeg = Osszegszamitas(s,n);
   atlag = Atlagszamitas(s,n);
   cout << "összeg: " << osszeg << endl;
   cout << "átlag : " << atlag << endl;
}
void Olvas(vekt x, int *db)
{
 int i;
  cout << "Az adatok száma: "; cin >> *db;
  for(i = 0; i<*db; i++)
   {
     cout << i << ". adat: ";
     cin >> x[i];
   }
}
double Osszegszamitas( vekt x, int db)
{
 int i;
 double s = 0.0;
   for(i = 0; i<db; i++)
   {
     s += x[i];
   }
   return (s);
}
double Atlagszamitas( vekt x, int db)
{
  return(Osszegszamitas(x,db)/db);
}
```

A program futásának eredménye:

```
Az adatok száma: 4
0. adat: 1
1. adat: 2
2. adat: 3
3. adat: 4 összeg: 10
átlag : 2.5
```

Művelet végzésére használjuk fel az alábbi struktúrának a típusdefinícióját függvényparaméterként:

```
typedef struct Muveletvegzes{
                              float x,y;
                              float eredmeny;
                              char  muv;
                             } MUVELET;
```

Írjunk *Olvas* függvényt a struktúra feltöltésére és *Szamol* függvényt a művelet elvégzésére.

```
/* muv_str.cpp */
#include <iostream.h>
typedef struct Muveletvegzes{
                              float x,y;
                              float eredmeny;
                              char  muv;
                             } MUVELET;
void Olvas(MUVELET *sz);
float Szamol(MUVELET *sz);

void main()
{
 MUVELET z;
  Olvas(&z);
  cout << z.x << " " << z.muv << " " << z.y << " = " << Szamol(&z);
}
void Olvas(MUVELET *sz)
{
  cout << "művelet (+,-,*,/): "; cin >> sz->muv;
  cout << "1. adat: "; cin >> sz->x ;
  cout << "2. adat: "; cin >> sz->y;
}
float Szamol(MUVELET *sz)
{
  sz->eredmeny = 0;
  switch (sz->muv)
 {
   case '+' : sz->eredmeny = sz->x+sz->y; break;
   case '-' : sz->eredmeny = sz->x-sz->y; break;
   case '*' : sz->eredmeny = sz->x*sz->y; break;
   case '/' : sz->eredmeny = sz->x/sz->y; break;
 }
return (sz->eredmeny);
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 4
2. adat: 5
4 * 5 = 20
```

1.16. Fájl kezelése

A C nyelvben lévő fájlműveletekhez hasonlóan a C++ is támogatja a szöveges és bináris fájl kezelését. A C++ rendelkezik egy fájlfolyam-gyűjteménnyel, amelynek használata lehetővé teszi a fájl beviteli és kiviteli (I/O) műveletek könnyű használatát.

Míg a *cin* bemeneti adatfolyam (*istream* objektum) és a *cout* kimeneti adatfolyam (*ostream* objektum) már előre deklarált volt, addig a fájl használatához a fájl objektumot deklarálni kell. Az *iostream.h* fejléc (*include*) fájl definiálja a *cout* és *cin* adatfolyamot. Hasonlóan a *fstream.h* fejléc fájl definiálja az *ofstream* kimeneti adatfolyam osztályt, melynek objektumai segítségével a programban fájlkivitelt hajthatunk vége. Az *fstream.h* fejléc fájlban definiált *ifstream* bemeneti adatfolyam osztály objektumaival fájlból olvashatunk. A fájl olvasása esetén az objektumnak *ofstream* típusúnak kell lennie.

1.16.1 Karakter írása és olvasása

Karakterenként olvashatunk a bemeneti fájlból a *get* és írhatunk kimeneti fájlba a *put* függvénnyel. A programba az *fstream.h* fejléc fájlt kell beszerkesztenünk. Tekintsük az alábbi deklarációt:

```
#include <fstream.h>
      ...
char kar;
ofstream FileIr;
ifstream FileOlvas;
```

Fájl nyitása írásra:

```
FileÍr.open("adatok.dat");
```

Írásra megnyitott fájlba egy karakter írása:

```
FileIr.put(kar);
```

Fájl megnyitása olvasásra:

```
FileOlvas.open("adatok.dat");
```

Olvasásra megnyitott fájlból egy karakter olvasása:

```
FileOlvas.get(kar);
```

Néha szükségünk van arra, hogy a következő karaktert csak vizsgálat céljából olvassuk be. Ha nincs szükségünk rá, lehetőség van a beolvasott karaktert visszatenni az kimeneti *stream*-re.

```
FileOlvas.putback(kar)
```

A fájl bezárása, akár írása, akár olvasásra volt megnyitva:

```
FileIr.close();

FileOlvas.close();
```

Olvassunk karakterenként a MONDAT.DAT fájlból és írjuk ki a beolvasott karaktert a KIIR.DAT fájlba. Az adat végét a pont (.) jelzi. A pont helyett felkiáltójelet írjunk ki a KIIR.DAT fájlba.

Addig olvasunk karakterenként, míg a *kar* változóba pont nem kerül, majd a *kar* tartalmát visszatesszük az input adatfolyamra a *putback* segítségével.

```
finp.get(kar);
while ( kar!= '.')
{
  fout.put(kar);
  finp.get(kar);
}
finp.putback(kar);
```

A feladat megoldása:

```
/* ch_io2.cpp */
#include <iostream.h>
#include <fstream.h>
void main()
{
 char kar;
 ifstream finp;
 ofstream fout;
  finp.open("mondat.dat");
  fout.open("kiir.dat");
  finp.get(kar);
  while (kar != '.')
  {
    fout.put(kar);
    finp.get(kar);
  }
  finp.putback(kar);
  fout.put('!');
  finp.close();
  fout.close();
}
```

A program futásának eredménye: a KIIR.DAT fájl.

A MONDAT.DAT fájlból olvas:

```
Ma szep az ido.
```

A KIIR.DAT fájlba ír:

```
Ma szep az ido!
```

A SZOVEG.DAT fájl tartalmát olvassuk a fájl végéig karakterenként és másoljuk át az ATIR.DAT fájlba.

Ha megnyitunk vagy létrehozunk egy fájlt, akkor a program biztonságos működése érdekében érdemes a művelet eredményét vizsgálni a fájlobjektum ***fail*** tagfüggvényével. Ha a fájlművelet alatt nem volt hiba, akkor a ***fail*** függvény hamis (0) értéket, hiba esetén igaz értéket ad vissza. Példaként nézzük meg a SZOVEG.DAT fájlnyitásának ellenőrzését:

```
finp.open("szoveg.dat");
if( finp.fail())
  {
    cout << "szoveg.dat input file nem létezik!\n";
    exit(1);
  }
```

A fájl végének vizsgálatához az ***eof*** tagfüggvényt használjuk. A függvény 0-át ad vissza, ha még nem értük el a fájl végét, különben 1-et. A vizsgálathoz a **while** ciklust használtuk fel:

```
  while(!finp.eof())
  {
     ...
  }
```

```
/* ch_io4.cpp */
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>

void main()
{
 char kar;
 ifstream finp;
 ofstream fout;
  finp.open("szoveg.dat");
```

```
    if( finp.fail())
   {
     cout << "szoveg.dat input file nem létezik!\n";
     exit(1);
   }
   fout.open("atir.dat");
   if(fout.fail())
   {
     cout << "atir.dat output file nyitási hiba!\n";
     exit(1);
   }
   finp.get(kar);
   while(!finp.eof())
   {
     fout.put(kar);
     finp.get(kar);
   }
   finp.close();
   fout.close();
}
```

A program futásának eredménye:

A SZOVEG.DAT fájlból olvas:

```
Ma szep az ido, de holnapra
el is romolhat.
```

Az ATIR.DAT fájlba ír:

```
Ma szep az ido, de holnapra
el is romolhat.
```

1.16.2. Különféle típusú adatok fájlkezelése

A különféle adatok fájlba való be- illetve kivitele az alábbi módon történik:

- kimeneti fájlfolyam használatával a beszúrás műveleti jellel (<<) írhatunk információt fájlba,
- bemeneti fájlfolyam használatával pedig a kiemelés műveleti jellel (>>) olvashatunk fájlban tárolt információt.

Az ADAT.DAT fájl három valós számot tartalmaz. Olvassuk be az adatokat és számítsuk ki az adatok átlagát, és azt az ERED.DAT fájlba írjuk ki.

```
/* file1_pg.cpp */
#include <fstream.h>
void main()
{
  ifstream inp;
  ofstream out;
  double a1,a2,a3;
  inp.open("adat.dat");
  out.open("ered.dat");
  inp >> a1 >> a2 >> a3;
  out << "Az adat.dat tartalmának átlaga: "
      << (a1+a2+a3)/3.0 << endl;
  inp.close();
  out.close();
}
```

A program futásának eredménye:

Az ADAT.DAT fájlból olvas:

```
2.1
3.5
4.2
```

Az ERED.DAT fájlba ír:

```
Az adat.dat tartalmának átlaga: 3.266667
```

Az ADAT.DAT fájl három valós számot tartalmaz. Olvassuk be az adatokat és számítsuk ki az adatok átlagát, és azt az ERED.DAT fájlba írjuk ki 3 tizedesjegyre.

```
/* file2_pg.cpp */
#include <fstream.h>
void main()
{
 ifstream inp;
 ofstream out;
 double a1,a2,a3;
  inp.open("adat1.dat");
  out.open("ered1.dat");
  out.setf(ios::fixed);
  out.setf(ios::showpoint);
  out.precision(3);
  inp >> a1 >> a2 >> a3;
  out << "Az adat1.dat tartalmának átlaga: "
      << (a1+a2+a3)/3.0 << endl;
  inp.close();
  out.close();
}
```

A program futásának eredménye:

Az ADAT1.DAT fájlból olvas:

```
2.1 3.5 4.2
```

Az ERED1.DAT fájlba ír:

```
Az adat1.dat tartalmának átlaga: 3.267
```

Az ADAT.DAT fájl három valós számot tartalmaz. Olvassuk be az adatokat és számítsuk ki az adatok átlagát, és azt az ERED.DAT fájlba írjuk ki 3 tizedesjegyre. Ellenőrizzük a fájlok nyitását és hiba esetén adjunk hibajelzést.

```
/* file3_pg.cpp */
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main()
{
 ifstream inp;
 ofstream out;
 double a1,a2,a3;
  inp.open("adat2.dat");
  if (inp.fail())
  {
    cout << "adat2.dat nem létezik\n";
    exit(1);
  }
  out.open("ered2.dat");
  if (out.fail())
  {
    cout << "ered2.dat létrehozás nem sikerült\n";
    exit(1);
  }
  out.setf(ios::fixed);
  out.setf(ios::showpoint);
  out.precision(3);
  inp >> a1 >> a2 >> a3;
  out << "Az adat2.dat tartalmának átlaga: "
      << (a1+a2+a3)/3.0 << endl;
  inp.close();
  out.close();
}
```

A program futásának eredménye:

Az ADAT2.DAT fájlból olvas:

```
2.1 3.5 4.2
```

Az ERED2.DAT fájlba ír:

```
Az adat2.dat tartalmának átlaga: 3.267
```

A három valós számot tartalmazó bemeneti fájl nevét és az eredményt tároló kimeneti fájl nevét olvassuk be billentyűzetről. Ellenőrizzük a fájlok nyitását és hiba esetén adjunk hibajelzést. Olvassuk be az adatokat és számítsuk ki az adatok átlagát, melyet 3 tizedesjegyre írjunk ki.

```
/* file4_pg.cpp */
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
void main()
{
 ifstream inp;
 ofstream out;
 float a1,a2,a3;
 char  InpFileName[14], OutFileName[14];
  cout << "File neve: "; cin >> InpFileName;
  inp.open(InpFileName);
  if (inp.fail())
  {
    cout << InpFileName << " nem létezik\n";
    exit(1);
  }
  cout << "Output file neve: "; cin >> OutFileName;
  out.setf(ios::fixed);
  out.setf(ios::showpoint);
  out.open(OutFileName);
  if (out.fail())
  {
    cout << OutFileName << " létrehozás nem sikerült\n";
    exit(1);
  }
  out.precision(3);
  inp >> a1 >> a2 >> a3;
  out << "Az adat.dat tartalmának átlaga: " << (a1+a2+a3)/3.0 << endl;
  inp.close();
  out.close();
}
```

A program futásának eredménye:

```
File neve: adat.dat
Output file neve: ered4.dat
```

Az ADAT.DAT fájlból olvas:

```
2.1
3.5
4.2
```

Az ERED4.DAT fájlba ír:

```
Az adat.dat tartalmának átlaga: 3.267
```

A három valós számot tartalmazó bemeneti fájl nevét és az eredményt tároló kimeneti fájl nevét olvassuk be billentyűzetről. Ellenőrizzük a fájlok nyitását és hiba esetén adjunk hibajelzést.Az adatok beolvasását, az átlagszámítást és az eredmény 3 tizedesjegyre való kiírását az *atlagszamitas* függvény végezze el.

```
/* file5_pg.cpp */
#include <iostream.h>
#include <fstream.h>
#include <iomanip.h>
#include <stdlib.h>
double atlagszamitas(istream& input, ofstream& output,
                     int mezoszelesseg, int tizedespont);
void main()
{
 ifstream inp;
 ofstream out;
 char  InpFileName[14], OutFileName[14];
  cout << "File neve: "; cin >> InpFileName;
  inp.open(InpFileName);
  if (inp.fail())
  {
    cout << InpFileName << " nem létezik\n";
    exit(1);
  }
  cout << "Output file neve: "; cin >> OutFileName;
  out.open(OutFileName);
  if (out.fail())
  {
    cout << OutFileName << " létrehozás nem sikerült\n";
    exit(1);
  }
  cout << "átlag : " << atlagszamitas(inp, out, 10, 3);
  inp.close();
  out.close();
}

double atlagszamitas(istream& input, ofstream& output,
                     int mezoszelesseg, int tizedespont)
{
  output.setf(ios::fixed);
  output.setf(ios::showpoint);
  output.setf(ios::showpos);
  output.precision(tizedespont);
  cout.setf(ios::fixed);
  cout.setf(ios::showpoint);
  cout.setf(ios::showpos);
  cout.precision(tizedespont);
  double adat, osszeg = 0.0;
  int db = 0;
```

```
  while ( input >> adat)
  {
    cout << setw(mezoszelesseg) << adat << endl;
    output << setw(mezoszelesseg) << adat << endl;
    db++;
    osszeg += adat;
  }
    output << "átlag : " << osszeg/db << endl;
  return (osszeg/db);
}
```

A program futásának eredménye:

```
File neve: adat.dat
Output file neve: ered5.dat
    +2.100
    +3.500
    +4.200
átlag : +3.267
```

Az ERED5.DAT fájlba ír:

```
    +2.100
    +3.500
    +4.200
átlag : +3.267
```

1.16.3. String fájlkezelése

Írjunk ki a beszúrás műveleti jellel 3 olyan sort, amely mindegyike legalább egy szóközt tartalmaz a SORINF.DAT fájlba.

```
/* sor_ir.cpp */
#include <iostream.h>
#include <fstream.h>
void main()
{
 ofstream out;
 cout << "A sorinf.dat fájlba 3 sort ir ki!" << endl;
 out.open("sorinf.dat");
 out << "Az 1. sor. " << endl;
 out << "A  2. sor. " << endl;
 out << "A  3. sor. " << endl;
 out. close();
}
```

A program futásának eredménye:

```
A sorinf.dat fájlba 3 sort ir ki!
```

A SORINF.DAT fájlból olvassuk be a 3 sort a kiemelés műveleti jellel.

```
/* sor_olv1.cpp */
#include <iostream.h>
#include <fstream.h>
void main()
{
 char sor1[80], sor2[80], sor3[80];
 ifstream in;
 cout << "A sorinf.dat fájlból 3 sort olvas!" << endl;
 in.open("sorinf.dat");
 in >> sor1;
 in >> sor2;
 in >> sor3;
 in. close();
 cout << sor1 << endl;
 cout << sor2 << endl;
 cout << sor3 << endl;
}
```

A program futásának eredménye:

```
A sorinf.dat fájlból 3 sort olvas!
Az
1.
sor.
```

A program eredményét kiértékelve nem a várt eredményt kaptuk, mivel a fájl bementi adatfolyam a *cin* -hez hasonlóan az elválasztó (jelen esetben a szóköz) karakterig olvasott, így már érthető, hogy a három sorban az első sor szóközzel elválasztott karaktersorozata jelent meg.

Az előző feladatot oldjuk meg a ***getline*** használatával.

A ***getline*** tagfüggvény a sor végéig ('\n') olvas, de a beolvasott karaktersorozat nem tartalmazza a sort termináló ('\n) karaktert. A függvény második paramétere a sztring hossza.

```
ifstream& getline(char*, int, char = '\n');
ifstream& getline(signed char*, int, char = '\n');
ifstream& getline(unsigned char*, int, char = '\n');
```

```
/* sor_olv2.cpp */
#include <iostream.h>
#include <fstream.h>
void main()
{
 char sor1[80], sor2[80], sor3[80];
 ifstream in;
 cout << "A sorinf.dat fájlból 3 sort olvas!" << endl;
 in.open("sorinf.dat");
 in.getline( sor1, sizeof(sor1));
 in.getline( sor2, sizeof(sor2));
 in.getline( sor3, sizeof(sor3));
 in. close();
 cout << sor1 << endl;
 cout << sor2 << endl;
 cout << sor3 << endl;
}
```

A program futásának eredménye:

```
A sorinf.dat fájlból 3 sort olvas!
Az 1. sor.
A 2. sor.
A 3. sor.
```

Amint az eredményből is látható, a ***getline*** függvénnyel szóközt tartalmazó karaktersorozat is olvasható.

Olvassuk be a SORINF.DAT fájlt a ***getline*** függvénnyel oly módon, hogy az olvasásnál figyeljük a fájlvége jelet.

```
/* sor_olv3.cpp */
#include <iostream.h>
#include <fstream.h>
void main()
{
 char sor[40][80];
 int  i = 0;
 ifstream in;
 cout << "A sorinf.dat fájlból 3 sort olvas!" << endl;
 in.open("sorinf.dat");
 while (!in.eof())
 {
   in.getline( sor[i], sizeof(sor[i]));
   i++;
 }
 in. close();
 cout << sor[0] << endl;
 cout << sor[1] << endl;
```

```
 cout << sor[2] << endl;
}
```

A program futásának ereménye:

```
A sorinf.dat fájlból 3 sort olvas!
Az 1. sor.
A  2. sor.
A  3. sor.
```

1.16.4. Az ios osztály használata

A 1.17. táblázat az *ios* megnyitási módjait tartalmazza.

1.17. táblázat

Nyitási mód	Hatása
ios::app	Megnyitja a fájlt (*append*) hozzáfűzés módban és a fájlmutatót a fájl végére helyezi.
ios::ate	A fájl mutatót a fájl végére helyezi.
ios::in	A fájlt (*input*) bevitelre nyitja meg, *ifstream* esetén ez az alapértelmezés.
ios::out	A fájlt (*output*) kimenetre nyitja meg, *ofstream* esetén ez az alapértelmezés.
ios::binary	A fájlt bináris módban nyitja meg.
ios::trunc	Ha a fájl nem létezik, akkor létrehozza, ha létezik megnyitja és felülírja.
ios::nocreate	Ha megadott fájl nem létezik, akkor nyitási hibát jelez.
ios::noreplace	Ha a fájl létezik, akkor nyitási művelet hibát jelez, kivéve *ate* vagy *app* van beállítva.

Ha például az *fnev* fájlt az alábbi nyitási módokkal nyitjuk meg

```
out.open (fnev, ios::out| ios::noreplace);
```

akkor megakadályozzuk a meglévő fájl felülírását.

Adott darabszámú egész típusú adatot írjunk fájlba, ezután a lezárt fájlhoz fűzzünk hozzá egy újabb egész típusú adatot, majd írassuk vissza a fájl tartalmát.

Az *ios* osztály *app* nyitási módját használva lehetőség van egy már létező fájl információjának bővítésére, vagyis a meglévő információhoz további adatok hozzáfűzésére.

Az *fnev* fájl megnyitása hozzáfűzésre:

```
    out.open(fnev, ios::app);
```

```
/* fileapp.cpp */
#include <fstream.h>
#include <iostream.h>
void main()
{
 int       db;
 int       adat;
 char      fnev[20];
 ifstream in;
 ofstream out;
   cout << "Az adat fájl neve: ";
   cin >> fnev;
   out.open(fnev);
   cout << "Adatok száma: ";
   cin >> db;
   for(int i = 0; i<db; i++)
   {
     cout << "Adat: "; cin >> adat;
     out << adat << endl;
   }
   out.close();
   cout << "Uj adat: "; cin >> adat;
   out.open(fnev, ios::app);
   out << adat;
   out.close();
   in.open(fnev);
   cout << fnev << " fájl tartalma: ";
   while (!in.eof())
   {
     in >> adat;
     cout << adat << " ";
   }
   in.close();
}
```

A program futásának eredménye:

```
Az adat fájl neve: app.dat
Adatok száma: 4
Adat: 6
Adat: 2
Adat: 1
Adat: 3
Uj adat: 12
app.dat fájl tartalma: 6 2 1 3 12
```

1.16.5. Bináris filekezelés

Ha a programban összetettebb adattípusokat, tömböt és struktúrákat kell írnunk fájlba és olvasnunk fájlból., akkor ezekhez a műveletekhez a *write* és a *read* tagfüggvényeket használhatjuk.

ostream::write tagfüggvénye:

```
ostream& write(const signed char*, int n);
ostream& write(const unsigned char*, int n);
```

istream::read tagfüggvénye:

```
istream& read(signed char*, int);
istream& read(unsigned char*, int);
```

Írjunk programot, amely az alábbi típusú struktúra adatmezőit TT.DAT fájlba írja.

```
typedef struct{
                  char neve[20];
                  int kora;
                  float atlaga;
               } TANULO;
```

```
/* bin_ir.cpp */
#include <iostream.h>
#include <fstream.h>
#include <string.h>
typedef struct{
                  char neve[20];
                  int kora;
                  float atlaga;
               } TANULO;

void main()
{
  TANULO t;
  ofstream out_tanulo;
  strcpy(t.neve,"Zoli");
  t.kora = 16;
  t.atlaga = 4.5;
  cout << "Tanuló neve   : " << t.neve << endl;
  cout << "       kora   : " << t.kora << " ,ves" << endl;
  cout << "       átlaga: " << t.atlaga << endl;
```

```
  cout << "A tt.dat fájlba kiirva." << endl;
  out_tanulo.open("tt.dat");
  out_tanulo.write((char*)&t, sizeof(TANULO));
  out_tanulo.close();
}
```

A program futásának ereménye:

```
Tanuló neve   : Zoli
       kora   : 16 éves
       átlaga: 4.5
A tt.dat fájlba kiirva.
```

Írjunk programot, amely a TT.DAT fájlt beolvassa!

```
/* bin_olv.cpp */
#include <iostream.h>
#include <fstream.h>
#include <string.h>
typedef struct{
                 char neve[20];
                 int kora;
                 float atlaga;
              } TANULO;

void main()
{
  TANULO t;
  ifstream in_tanulo;
  in_tanulo.open("tt.dat");
  in_tanulo.read((char*)&t, sizeof(TANULO));
  cout << "A tt.dat fájl tartalma: " << endl;
  cout << "Tanuló neve   : " << t.neve << endl;
  cout << "       kora   : " << t.kora << " ,ves" << endl;
  cout << "       átlaga: " << t.atlaga << endl;
  in_tanulo.close();
}
```

A program futásának eredménye:

```
A tt.dat fájl tartalma:
Tanuló neve   : Zoli
       kora   : 16 éves
       átlaga: 4.5
```

Módosítsuk a BIN_IR.CPP programot, amellyel néhány tanuló adatát tároljuk a TANULOK.DAT fájlba!

```
/* bin_ir2.cpp */
#include <iostream.h>
#include <fstream.h>
#include <string.h>
typedef struct{
                  char neve[20];
                  int kora;
                  float atlaga;
               } TANULO;

void main()
  TANULO t;
  int    i,n;
  ofstream out_tanulo;
  cout << "A tanulók száma: "; cin >> n;
  out_tanulo.open("tanulok.dat");
  cout << "A tanulók adatai: " << endl;
  for(i = 0; i<n; i++)
  {
    cout << "\nA tanuló neve  : "; cin >> t.neve;
    cout << "A tanuló kora  : "; cin >> t.kora;
    cout << "A tanuló átlaga: "; cin >> t.atlaga;
    out_tanulo.write((char*)&t, sizeof(TANULO));
    cout << endl;
  }
  out_tanulo.close();
}
```

A program futásának eredménye:

```
A tanulók száma: 5
A tanulók adatai:

A tanuló neve  : Zoli
A tanuló kora  : 12
A tanuló átlaga: 4.5

A tanuló neve  : Kati
A tanuló kora  : 10
A tanuló átlaga: 3.4

A tanuló neve  : Szilvi
A tanuló kora  : 9
A tanuló átlaga: 4.2

A tanuló neve  : Endre
A tanuló kora  : 14
A tanuló átlaga: 2.9
```

```
A tanuló neve   : Anna
A tanuló kora   : 16
A tanuló átlaga: 4.4
```

A TANULOK.DAT fáj tartalmának készítsük el a listáját, majd az első és a negyedik tanuló adatait írassuk ki. A feladat megoldáshoz használjuk a ***tellg*** és a ***seekg*** tagfüggvényeket!

.A ***tellg*** tagfüggvény az aktuális fájlpozíciót adja vissza:

```
long tellg();
```

A ***seekg*** tagfüggvény a fájlban az aktuális pozíciót áthelyezi az *offset* paraméterben megadott bájttal, a *dir* pedig megadja az *offset* áthelyezési irányát.

```
istream& seekg(streamoff offset, seek_dir dir);
```

Az irány a *seek_dir* sorszámozott típusban van definiálva:

```
enum seek_dir(beg,cut,end);
```

```
/* file_bin2.cpp */
#include <iostream.h>
#include <fstream.h>
#include <string.h>
typedef struct{
                    char neve[20];
                    int kora;
                    float atlaga;
                 } TANULO;

void main()
{
  TANULO t[20], t1, t2;
  int i = 0;
  streamoff k;
  ifstream in_tanulo;
  in_tanulo.open("tanulok.dat");
  in_tanulo.read((char*)&t[i], sizeof(TANULO));
  while(!in_tanulo.eof())
  {
    cout << "Tanuló neve   : " << t[i].neve  << endl;
    cout << "       kora   : " << t[i].kora << " éves" << endl;
    cout << "       átlaga: " << t[i].atlaga << endl;
    i++;
    in_tanulo.read((char*)&t[i], sizeof(TANULO));
  }
  in_tanulo.close();
```

```
    in_tanulo.open("tanulok.dat");
    k =  in_tanulo.tellg();
    cout << "\nAz aktuális pozíció: " << k;
    in_tanulo.seekg(k);
    in_tanulo.read((char*)&t1, sizeof(TANULO));
    cout << "\nTanuló neve   : " << t1.neve   << endl;
    cout << "         kora   : " << t1.kora << " éves" << endl;
    cout << "         átlaga: " << t1.atlaga << endl;

    in_tanulo.seekg(2L*sizeof(TANULO)+1,ios::cur);
    in_tanulo.read((char*)&t2, sizeof(TANULO));
    cout << "\nTanuló neve   : " << t2.neve   << endl;
    cout << "         kora   : " << t2.kora << " éves" << endl;
    cout << "         átlaga: " << t2.atlaga << endl;

    in_tanulo.close();
}
```

A program futásának eredménye:

```
Tanuló neve   : Zoli
       kora   : 12 éves
       átlaga: 4.5
Tanuló neve   : Kati
       kora   : 10 éves
       átlaga: 3.4
Tanuló neve   : Szilvi
       kora   : 9 éves
       átlaga: 4.2
Tanuló neve   : Endre
       kora   : 14 éves
       átlaga: 2.9
Tanuló neve   : Anna
       kora   : 16 éves
       átlaga: 4.4

Az aktuális pozíció: 0
Tanuló neve   : Zoli
       kora   : 12 éves
       átlaga: 4.5

Tanuló neve   : Endre
       kora   : 14 éves
       átlaga: 2.9
```

1.17. A main függvény

Minden C(++) program kötelezően tartalmaz egy ***main*** nevű függvényt. Definiálásának helye lényegtelen, akármelyik forrásfájl tartalmazhatja tetszőleges elhelyezésben, a vezérlést a program indulásakor – a megfelelő inicializáló rendszerrutin (*startup* kód) lefutása után – mindenképp a ***main*** kapja meg.

A ***main*** visszatérési típusát kétféleképpen is meg lehet adni: **int**-nek és **void**-nak deklarálva. (Ne feledjük el, ha nem adunk meg típust a definícióban, a fordító automatikusan **int**-et tételez fel.) A visszaadott érték, ha van, a program ún. státuszkódja lesz. Ezt a program futása után megvizsgálhatjuk a DOS-ban az ERRORLEVEL *bacth* funkcióval.

Amennyiben programunkat nem a COMMAND.COM indította el, hanem egy másik felhasználói program hívta (lásd a **spawn** és **exec** függvénycsaládot a *process.h* fejléc fájlban), akkor az megkapja a visszaadott státuszkódot és felhasználhatja annak eldöntésére, hogy programunk sikeresen futott le vagy sem. Megállapodás szerint a 0 státuszkód sikeres végrehajtást jelent, az ettől eltérő értékekkel pedig a hiba jellegét is lehet közölni (például fatális hiba, fájlkezelési hiba, *Ctrl Break*, stb.). Minden komolyabb programnak illik visszadni státuszkódot – erre legtöbbször az **exit** függvényt használjuk (melyet szintén a *process.h* fájlban definiáltak). Ennek előnye, hogy a hibát észlelő bármely rutinból meghívható az exit, nem szükséges a vezérlést – a hibainformáció cipelésével – visszajuttatni a main-hez, bár sokszor elég nehéz megtalálni egy eldugott **exit** hívást.

A ***main*** meghívásakor három paramétert kap, de ezek közül csak annyit kell átvennie, amennyire ténylegesen szüksége van. Elhagyni azonban csak hátulról előre haladva lehet a paraméterlistáról. A ***main*** teljes deklarációja:

```
int main (int argc, char *argv[], char *env[]);
```

Az *argc* paraméter megadja a parancssorban átadott argumentumok számát. Az *argv[]* az egyes paramétereket tartalmazó sztringekre mutató pointerek tömbje, az *env[]* pedig az ún. környezeti változókat (*current variables*) és értéküket tartalmazó sztringek mutatótömbje.
Ha a programunk neve PROG, akkor a

SET KV = ALMA
PROG alfa beta GAMMA

DOS parancssorok hatására a PROG-ban lévő paraméterei így alakulnak:

- *argc* értéke 4
- *argv[0]* a PROG teljes elérési útjára mutat,
- *argv[1]* az "alfa' sztringre mutat,
- *argv[2]* a "beta" sztringre mutat,
- *argv[3]* a "GAMMA" sztringre mutat,
- *argv[4]* értéke NULL,
- *env[n]* a "KV = ALMA" sztringre mutat.

Az *n* értéke függ attól, hogy milyen környezeti változók voltak már korábban definiálva. Az *env* tömb végét szintén a NULL értékű elem jelzi.

A környezeti változók megadásának általános alakja:

SET ENVAR = *value*

- ENVAR a környezeti változó neve (például PATH vagy COMSPEC).
- value értéke az ENVAR beállításától függ, PATH esetén például lehet C:\DATA vagy COMSPEC esetén C:\DOS\COMMAND.COM

Írjunk olyan programot, amely a program indításakor megadott két fájl nevét paraméterként veszi át. Az első fájlból karakterenként olvasunk, a kisbetűket alakítsuk át nagybetűkké és írjuk ki a másodikként megadott fájlba.
A feladat megoldását a PRG_MAIN.CPP állomány tartalmazza.

A SZOVEG.DAT tartalma:

```
Ez egy proba adat az islower es toupper hasznalatara.
```

```
/* prg_main.cpp */
#include <iostream.h>
#include <fstream.h>
#include <string.h>
#include <ctype.h>
#include <process.h>
int main(int argc, char *argv[])
{
  ifstream inp;
  ofstream out;
  char kar;
  char InpFileName[13], OutFileName[13];
  if( argc <=2 ) cout << "Hiányos paraméterek!";
```

```
 else
  {
    strcpy(InpFileName, argv[1]);
    strcpy(OutFileName, argv[2]);
    inp.open(InpFileName);
    if( inp.fail())
    {
      cout << InpFileName << " nem létezik!" << endl;
      exit(1);
    }
    out.open(OutFileName);
    if( out.fail())
    {
      cout << OutFileName << " nem létezik!" << endl;
      exit(2);
    }

    while (!inp.eof())
    {
      inp.get(kar);
      if( kar != ' ' && kar != '.')
      {
        if(islower(kar)) kar = toupper(kar);
      }
      cout << kar;
      out.put(kar);
    }
    cout << endl;
    inp.close();
    out.close();
  }
  return 0;
}
```

Indítsuk el a programot az alábbi módon:

```
prg_main szoveg.dat atir.dat
```

A program futásának eredménye:

EZ EGY PROBA ADAT AZ ISLOWER ES TOUPPER HASZNALATARA.

Az eredmény a képernyőre és a ATIR.DAT fájlba is kiíródik.

Az ATIR.DAT tartalma:

EZ EGY PROBA ADAT AŽ ISLOWER ES TOUPPER HASZNALATARA.

1.18. Az előfeldolgozó

Minden C(++) fordítóprogram szerves részét képezi az ún. előfeldolgozó (preprocessor). A fordítóprogram és a preprocesszor kapcsolatát úgy kell elképzelni, hogy a tulajdonképpeni fordító nem is látja a forrásszöveget, hanem csak azt, amit ebből az előfeldolgozó készít a számára. A belső fordító nem ismeri a megjegyzések szintaxisát, ugyanis nincs rá szüksége: a neki átadott sorokból az előfeldolgozó azokat "kiírtja". A Borland C++ integrált fejlesztői rendszerénél a két rész kapcsolata olyan szoros, hogy nincs is lehetőség az előfeldolgozó kimenetének a megtekintésére.

Az előfeldolgozó egy sororientált, szövegfeldolgozó (más szóval makrónyelv), ami semmit sem "tud" a C(++) nyelvről. Ez két fontos követelménnyel jár: az előfeldolgozónak szóló utasításokat nem írhatjuk olyan kötetlen formában, mint az egyéb C(++) utasításokat (tehát egy sorba csak egy utasítás kerülhet és a parancsok nem lóghatnak át a másik sorba, hacsak nem jelöljük ki folytatósornak); másrészt minden, amit az előfeldolgozó művel, szigorúan csak szövegmanipuláció, függetlenül attól, hogy C(++) nyelvi alapszavakon, kifejezéseken vagy változókon dolgozik. A preprocesszornak szóló parancsokat a sor elején (esetleg szóközök és/vagy tabulátorok után) álló # karakter jelzi.

1.18.1. Szimbólumok és makrók

Az előfeldolgozónak szintén lehetnek "változói", ezeket szimbólumoknak, illetve makróknak nevezzük, és ugyanaz a képzési szabály vonatkozik rájuk, mint más azonosítókra. Azért, hogy a preprocesszor számára definiált szimbólumok a C(++) forrásnyelvi szövegben élesen különváljanak a programban használt azonosítóktól, szokás szerint a szimbólumokat csupa nagybetűvel képezzük. Szimbólumokat a következő paranccsal hozhatunk létre.

```
#define szimbólum helyettesítendő szöveg
```

A három fő rész tetszőleges számú, minimum egy szóköz és/vagy tabulátor választja el. Az előfeldolgozó minden beérkező sort átvizsgál, tartalmaz-e helyettesítő karaktersorozatot, és újból átvizsgálja a sort szimbólumokat keresve, amit új helyettesítés követhet, stb. Mindaddig folytatódik ez a folyamat, amíg vagy nem talál a sorban szimbólumot, vagy csak olyat talál, ami már egyszer helyettesítve lett (a végtelen rekurziók elkerülésére).

Példák szimbólumdefinícióra:

```
#define    EOS         '\0'
#define    MENUCOL     20
#define    MENULINE    5
#define    BORDER      2
#define    MENUROW     (MENULINE+BORDER)
#define    MENUSIZE    (MENUROW*MENUCOL)
```

A fenti definíciók a szimbólumok leggyakrabban használt területét mutatják be: legtöbb hasznuk az, hogy a különböző "bűvkonstansok" névvel ellátva egy helyre gyűjthetjük össze, világosabbá téve használatukat és megkönnyítve módosításukat. Az utolsó példát annak illusztrálására hoztuk fel, hogy milyen fontos szem előtt tartani azt, hogy szöveghelyettesítésről van szó. Ha a látszólag felesleges zárójelpárt elhagynánk, akkor a következő sorból

```
#define    MENUSIZE    (MENUROW*MENUCOL)
```

az alábbi keletkezne

```
#define    MENUROW     MENULINE+BORDER*MENUCOL
```

ami végeredményben a kívánt 140 helyett 45-öt eredményezne mindenütt, ahol MENUSIZE-t használjuk. Mivel a felesleges zárójelek bajt nem okozhatnak, szokjuk meg, hogy minden definícióban szereplő kifejezést zárójelezünk.

A makrók lehetővé teszik azt, hogy a szöveghelyettesítés paraméterezhető legyen. A paraméterek számára nincs elvi korlát megadva. Példák makrodefinícióra:

```
#define    abs(x)      ((x) < 0 ? (-(x)) : (x))
#define    min(a,b)    ((a) < (b) ? (a) : (b))
```

Figyeljük meg, hogy – szintén a szöveghelyettesítést szem előtt tartva – makrók esetében nemcsak a helyettesítő kifejezés, de a helyettesítő paramétereket is zárójelekkel védjük. Az első példában szereplő makró alábbi alakú hívása:

```
alfa = abs(y+2);
```

így a következő sort eredményezi

```
alfa = ((y+2) < 0 ? (-(y+2)) : (y+2));
```

Ha *y* értéke -3, akkor alfa-ba 1 kerül, mint az várható. Ha azonban a definíció helyettesítő részében az *x*-et védő zárójeleket elhagytuk volna, akkor *alfa* 5-öt kapna értékül. Ítt hívjuk fel a figyelmet arra is, hogy alfa kiszámításához a fenti esetben a *y+2* értékét kétszer kell meghatározni. Egyszer a feltétel kiértékelésekor, aztán az eredmény meghatározásához. Ez komplikáltabb kifejezés esetében a program hatékonyságát jelentősen ronthatja. Nagyobb bajt okoz azonban az, hogy a C-ben bizonyos kifejezéseknek ún. mellékhatásuk is lehet (például a kifejezés tartalmazhatja olyan függvény meghívását, ami a kívánt érték kiszámítása mellett, azt ki is írja a képernyőre), és a többszöri kiértékelés a mellékhatás többszöri jelentkezését vonja maga után (példánknál a részeredmény kétszer kerül kiírásra). Az ilyen makrók használatánál ezért célszerűbb az adott kifejezést egy ideiglenes változóba helyezni, és azzal meghívni a makrót. A makrók használatának szintaxisa, mint majd látni fogjuk, megegyezik a függvények hívásának szintaxisával. Ez nem véletlen, ugyanis több könyvtári "függvényt" például sok C(++) rendszer makróként valósít meg, ilyen "függvény" a ***getchar***() is, ami a használatát nem befolyásolja – a fent említett kivételektől eltekintve, mint például ***toupper***(), ***tolower***(), ***min***(), ***max***() stb. – ugyanakkor a rutinhívások elmaradása miatt gyorsabb kódot eredményez. Felhasználhatók a makrók arra is, hogy a program egy új telepítésnél az adott rendszerben esetleg eltérő nevű, vagy paraméterezésű könyvtári függvényeket segítségükkel a korábbi alakban használhassuk.

Ha egy szimbólum vagy makró által lefoglalt azonosítót fel szeretnénk szabadítani, azt az

```
#undef szimbólum
```

utasítással tehetjük meg. Ezt követően az előfeldolgozó a szimbólum (makró) további előfordulásait nem kíséri figyelemmel. Természetesen egy újabb #define utasítással újra definiálhatjuk ezt a szimbólumot is.

Megjegyzések:

– A következő szimbólumok nem szerepelhetnek sem #define, sem #undef direktívákban, mert foglalt kulcsszavak:

  __STDC__   __FILE__   __LINE__
  __DATE__   __TIME__

- Két szintaktikai egységet (token-t) egybe lehet forrasztani egy makró definíciós részében, ha ## választja el őket egymástól (és bármelyik oldalon esetleg szóközök). Az előfeldolgozó eltávolítja az esetleges szóközöket és a ## jelet, és egyesíti a két különálló szintaktikai egységet. Ily módon lehet változókat konstruálni. Például a #define VAR(I,J) (I ## J) makródefiníció esetén a VAR(x,6) szövegből x6 lesz a kifejtés után.
- A makrók definíciós részében lévő beágyazott makrók kifejtése a külső makró kifejtésekor történik meg, nem pedig a definíciójakor. Ennek leginkább olyankor van jelentősége, amikor egy beágyazott makrót #undef direktívában is használunk.
- A # karakter elhelyezhető bármely makróargumentum előtt, ilyen módon azt sztringgé alakítva. A makró kifejtésekor a #arg alakú paraméter hivatkozásoknak az "arg" alakja helyettesítődik. Például a

```
#define PRINTVAL(value) printf(#value"=%d\n",value)
```

makródefiníció esetében a

```
int counter = 0;
. . .
PRINTVAL(counter);
```

programrészletből

```
int counter = 0;
...
printf("counter" "=%d\n",counter);
```

lesz. A fenti ***printf*** utasítás pedig ekvivalens a

```
printf("counter=%d\n",counter);
```

utasítással.
- Más implementációkkal ellentétben a Borland C++ előfeldolgozója nem végez paraméterhelyettesítést sztringeken és karakterállandókon belül.

1.18.2. Feltételes fordítás

A C(++) nyelvi feltételes fordítás lehetőségét szintén a preprocesszor biztosítja. Ennek általános formája:

```
#if  feltétel
     ...
#elif feltétel
     ...
#else
     ...
#endif
```

ahol a *feltétel* konstans értéke vagy igaz (nem nulla), vagy hamis (nulla). Az **#elif** ágból tetszőleges számú lehet, akár el is maradhat, csakúgy, mint az **#else** ág. Ha az első feltétel nem teljesül, akkor az előfeldolgozó a **#if** utasítást követő első **#elif**, **#else**, illetve **#endif** direktíváig terjedő sorokat figyelmen kívül hagyja – tehát ezekben szerepelhetnek akár szintaktikailag hibás C++ utasítások is –, és a következő **#elif** utasítást veszi hasonlóképpen, ha van. Ha egyik feltétel sem teljesül, és van **#elif** ág, akkor csak az abban szereplő sorok kerülnek további feldolgozásra. Ha valamelyik feltétel igaz volt, akkor az adott ágat követő első #**elif** vagy **#else** utasítástól – ha van ilyen – a megfelelő **#endif**-ig terjedő sorok lesznek "eldobva". A feltételes fordítású részek tetszőleges mélységben egymásba ágyazhatók, például:

```
#if ALFA < 3
     ...
#if defined(BETA)
     ...
#else
     ...
#endif
     ...
#elif ALFA < 10
     ...
#endif
```

A **defined** (szimbólum) alakú kifejezés akkor igaz, ha az adott szimbólum – tetszőleges értékkel – definiálva van, egyébként hamis. Az **#if defined** (szimbólum) a következő alakban is írható: **#ifdef** szimbólum, illetve a feltétel negáltjának megfelelően létezik **#ifndef** szimbólum alakú utasítás is.

1.18.3. Előredefiniált szabványos szimbólumok

A Borland C++ mind az 5, az ANSI által megkívánt előredefiniált szimbólumot rendelkezésre bocsátja. Mindegyik szimbólum két-két aláhúzás karakterrel kezdődik és végződik. Ezek a következők:

- __LINE__ Az aktuális forrásfájl feldolgozás alatt álló sorának sorszáma 1-ről indul.
- __FILE__ Az aktuális forrás- vagy fejléc fájl nevét tartalmazó sztringkonstans.
- __DATE__ Az a dátum, amikor az előfeldolgozó az aktuális forrásfájl feldolgozásához hozzákezdett, sztringkonstans formájában.
- __TIME__ Az az időpont, amikor az előfeldolgozó az aktuális forrásfájl feldolgozásához hozzákezdett, sztringkonstans formájában.
- __STDC__ A szimbólum 1 értékkel definiálva van, ha az *Ansi keyword only* fordításvezérlő kapcsoló On-ra van állítva (ilyenkor a fordító csak az ANSI által támogatott kulcsszavakat ismeri fel), egyébként a szimbólum definiálatlan.

1.18.4. A Borland C++ saját előredefiniált szimbólumai az 5.02 verzióban

1.18. táblázat

Makró	Értéke	Jelentése
__BCOPT__	1	Minden olyan fordítóban definiálva van, amelyben van optimalizáló.
__BCPLUSPLUS__	0x340	Definiálva van, ha C++ fordítást írtunk elő. Értéke későbbi verziókban növekedhet. Ez a fordító verzió számára utal.
__BORLANDC__	0x460	Verziószám. Későbbi verzióban növekedhet.
__CDECL__	1	Akkor van definiálva, ha a hívási konvenciót C-re állítottuk, egyébként definiálatlan.
__CHAR_UNSIGNED	1	Alapértelmezésben definiálva van, azt jelzi, hogy a char **unsigned** értelmezésű. A -K opcióval kikapcsolható.
__CONSOLE__		Csak a 32 bites fordítóban érhető el. Ha definiálva van, azt jelzi, hogy a program csak konzol alkalmazás.

Makró	Értéke	Jelentése
__cplusplus	1	C++ fordítási módban definiálva van, egyébként nem.
__DLL__	1	Definiálva van ha a program kódgenerálása (*prolog/epilog*) Windows DLL típusúra van állítva, egyébként nincs definiálva.
_M_IX86	1	Mindig definiálva van.
__MSDOS__	1	Egész konstans.
__MT__	1	Csak akkor van definiálva a 32 bites fordító számára, ha a -WM opciót használjuk. Ebben az esetben a többszálas (*multithread*) könyvtár kerül beszerkesztésre.
__OVERLAY__	1	A Borland C++ fordító családra jellemző opció. Előredefiniált értéke 1. Ha a -Y opcióval engedélyezzük az overlay fordítást. Ha nem engedélyezzük, akkor definiálatlan ez a makró.
__PASCAL__	1	Definiálva van, ha hívási konvenciót PASCAL típusúra állítottuk, egyébként definiálatlan.
__TCPLUSPLUS__	0x340	Verziószám.
__TEMPLATES__	1	A Borland C++ fordítókra jellemző. 1 értékkel van definiálva C++ fájlok fordítása esetén (azt jelenti, hogy a Borland C++ támogatja a templatek használatát), egyébként definiálatlan.
__TLS__	1	A 32 bites fordító használata esetén mindig 1 (igaz) értékű.
__TURBOC__	0x460	Verziószám, később növekedhet.
_WCHAR_T	1	Csak C++ programok esetében van definiálva, azt jelzi, hogy a **wchar_t** beépített adattípus.
_WCHAR_T_DEFINED	1	Csak C++ programok esetében van definiálva, azt jelzi, hogy a **wchar_t** beépített adattípus
_Windows		Windows 16 és 32 bites fordítások esetében definiálva van.
__WIN32__	1	A 32 bites fordító esetében mindig definiálva van, mind konzol applikációk, mind grafikus felhasználói felülettel rendelkező alkalmazások esetében.

1.18.5. Fájlbeépítés

Igen gyakran használjuk a preprocesszor fájlbeépítési (*file include*) képességét. Segítségével teljes fájlok építhetők be a forrásunkba, pontosan úgy, mintha ott szerepelnének beírva. Az ilyen beépítendő *include* fájlok egy helyen össze-

gyűjtve tartalmaznak szimbólum és makródefiníciókat, struktúrákat írnak le, más modulokkal definiált tárolási egységeket deklarálnak stb. A Borland C++ rendszer nagyszámú *include* fájlt bocsát a programozók rendelkezésére, amelyek közül az adott feladathoz szükségeseket építhetik be. A fejléc fájlok kiterjesztése hagyományosan .h (*header file*). Az *include* fájlokba beágyazva lehetnek újabb beépítési utasítások stb. A beépítési utasítás általános alakja:

#include <file-név>

vagy

#include " file-név "

vagy

#include szimbólum

A harmadik verzió esetében az *include* argumentumában sem a < jel sem " (idézőjel) nem szerepel az *include*-ot követően. A preprocesszor feltételezi, hogy a szimbólumot egy makróként definiáltuk és értéke az első két formátum szerinti fájlnév megadás.

1.18.6. Fordítási hibaüzenetek

```
#error HIBAÜZENET
```

Ez a direktíva az alábbi formátumú fordítási hibaüzenetet generálja:

```
Error: filename line# : Error directive: HIBAÜZENET
```

Ezt a direktívát általában olyan feltételes fordítási utasítások közé ágyazzuk be, amelyek segítségével nemkívánatos fordítási hibákat igyekszünk elkapni. Normális esetben a fordítási feltétel hamis lesz. Ha a hibafeltétel igaz, akkor azt szeretnénk, hogy a fordító egy hibaüzenetet nyomtasson és álljon meg. Ez a **#error** direktíva segítségével érhető el.

Például:

```
#if (SAJATSZIMBOLUM != 0 && SAJATSZIMBOLUM != 1)
#error SAJATSZIMBOLUM erteke 1 vagy 0 kell legyen
#endif
```

1.18.7. Implementáció-függő vezérlősorok

Ez az ún. pragma direktívák, amelyeknek általános alakja:

```
#pragma direktívanév paraméterek
```

Ezekben minden fordítónak lehetősége van arra, hogy tetszőleges, implementáció-függő vezérlőutasítást szabjon meg, ugyanis – definíció szerint – ha egy fordító nem ismer fel egy pragma direktívanevet, akkor az egész sort figyelmen kívül kell hagynia.

A Borland C++ 5.02-es verzió a következő **pragma** utasításokat ismeri:

```
#pragma argsused
```

Csak függvénydefiníciók között használható és csak a következő függvényre van hatása. Letiltja a következő hiba üzenetet:

Parameter név *is never used in function* függvénynév

```
#pragma anon_struct  on / off
```

Az *anon_struct* on direktíva lehetővé teszi, hogy osztályokba ágyazott anonim struktúrákat használjon.

```
#pragma codeseg <szegmens név> <osztály> <csoport>
```

Ezen direktíva segítségével megnevezhető az a szegmens név, osztály vagy csoport, amelyet egy függvényhez rendelünk. Paraméter nélkül használva ezt a direktívát, az alapértelmezés szerinti kódszegmens lesz felhasználva.

```
#pragma comment típus "sztring"
```

Ezen direktíva segítségével megjegyzés rekordok írathatók a kimeneti .OBJ állományba.
A típus a következő lehet:

exestr	A string az .OBJ fájlba kerül oly módon, hogy a programszerkesztő azt elhelyezi a futtatható állományban. A program futtatásakor a sztring nem töltődik be a memóriába, de a diszken keresőprogrammal megtalálható.
lib	A kommentként az .OBJ fájlba írt sztringet a programszerkesztő könyvtárkeresési útnak tekinti.
user	A fordító a sztringet a .OBJ fájlba írja, de a programszerkesztő ezt nem veszi figyelembe.

```
#pragma exit függvénynév <prioritás>
```

Ez a direktíva használható arra, hogy megadjuk annak a függvénynek a nevét, amelyiket a programból való kilépéskor, az _exit előtt aktivizálni kell. A prioritás 64-255 közötti érték kell legyen, a legnagyobb prioritás: 0. A nagyobb prioritású függvények később kerülnek meghívásra a programból való kilépéskor. Ha nem adjuk meg, a prioritás értéke 100. A 0 és 63 közötti prioritás közötti értékeket a C könyvtárak használják.

```
#pragma hdrfile "fájlnév.CSM"
```

Ezzel a direktívával adható meg annak a fájlnak a neve, amelyikben az előre lefordított fejléceket (*precompiled header*) tároljuk. Ha nem használunk előre lefordított fejléceket, akkor nincs hatása.

```
#pragma hdrstop
```

Ezzel a direktívával jelezhető az előrefordítandó fejlécfájlok listájának vége.

```
#pragma inline
```

Ez a direktíva megfelel a -B parancssor opciónak, vagy az integrált fejlesztői környezet (IDE) *inline* opciójának.

```
#pragma intrinsic [-]függvénynév
```

Ez a direktíva használható arra, hogy felülbíráljuk azokat a parancssor paramétereket vagy IDE opciókat, amelyek az *inline* függvény kezelésre vonatkoznak.

```
#pragma message szöveg
```

Ennek segítségével a fordítás során a felhasználó által megadott szöveg iratható a fordítási listába. Például:

```
#ifdef __BORLANDC__
#pragma message Ön a  BC++  __BORLANDC__ verzióját használja.
#else
#pragma message ("Ön nem a BC++ fordítót használja!")
#endif
```

Ez a második forma a Microsoft C fordítóval kompatibilis.

```
#pragma option [opciók ...]
```

Ezzel a pragma utasítással parancssor opciókat adhatunk meg forrásprogramon belül.

```
#pragma saveregs
```

Ez a **pragma** garantálja, hogy egy **huge** függvény nem változtatja meg a regiszterek értékét, amikor aktivizálásra kerül. Sokszor assembly nyelvű kód illesztésénél használjuk. A direktívát közvetlenül a függvénydefiníció előtt kell elhelyezni, és csak az őt követő függvényre hat.

```
#pragma startup függvénynév <prioritás>
```

Ez a direktíva használható arra, hogy megadjuk annak a függvénynek a nevét, amelyiket a programba való belépéskor a main előtt kell aktiválni. A prioritás 64-255 közötti érték kell legyen. A nagyobb prioritású függvények előbb kerülnek meghívásra a programba való belépéskor. Ha nem adjuk meg, a prioritás értéke 100. A 0 és 63 közötti prioritás közötti értékeket a C könyvtárak használják.

```
#pragma warn [+|-|.]www
```

Segítségével a figyelmeztetési szint állítható át. Például:

```
#pragma warn +xxx
#pragma warn -yyy
#pragma warn .zzz
```

A fenti utasítások hatására az *xxx* figyelmeztető üzenet generálását bekapcsoljuk, az *yyy* üzenetét kikapcsoljuk, míg a *zzz* üzenetre vonatkozó értelmezést az eredeti értékre állítjuk vissza. A használható hárombetűs kódok listája az integrált fejlesztői környezetben a help funkcióval (*command line options* tételnél) lekérhető.

2. C++ nyelv tulajdonságai

Ebben a fejezetben megismerkedünk azokkal a tulajdonságokkal, amivel a C++ nyelv kibővült: hivatkozás (v. referencia) típus, új operátorok és a függvények átdefiniálása.

2.1. C++ hivatkozás (referencia) használata

Míg a C nyelvben a függvényargumentumok érték szerint adódnak át, addig a C++ az argumentumokat érték szerint és hivatkozással is át lehet adni a függvényeknek. A C++ hivatkozási típus, amely a pointer típushoz szorosan kapcsolódik, az objektumokra alternatív típusokat hoz létre és a függvénynek hivatkozással adja át az argumentumot. Valójában tehát címszerinti paraméterátadás történik a függvénynek, de a függvényhíváskor nem pointer értéket kell az aktuális paraméterlistán elhelyezni; elegendő egy változót megadnunk. A fordítóprogram gondoskodik arról, hogy a változóra vonatkozó referencia helyettesítődjék be, ha a formális paraméterlista szerint az adott argumentum hivatkozás (referencia) típusú; hasonlóan a Pascal eljárások VAR paramétereihez. Míg a C nyelvben mutatók segítségével tudjuk a függvény paraméterek értékét a függvényben változtatni, addig a C++ nyelvben ez a referencia típus segítségével egyszerűbben megoldható.

2.1.1. Egyszerű hivatkozások

A hivatkozási típus felhasználásával már létező változókra hivatkozhatunk, alternatív nevet definiálva. A definíció formája:

```
típus& azonosító = objektum;
```

A referencia definiálásakor kötelező értéket adnunk:

```
int   k;

int&  n = k; // n a k változó alternatív (alias) neve
```

2.1.2. Hivatkozási típusú függvényparaméter

A C++ hivatkozások hivatkozások leegyszerűsítik a függvényparaméterek megváltoztatásának mechanizmusát, mert kiküszöbölik mutató- és a nem mutató változókat.

Példákon keresztül mutatjuk be a hivatkozási típusú paraméterek használatát.

```
/* ref_1.cpp */
#include <iostream.h>
void olvas( double *x1, double *x2);
void ref_olvas(double& w1, double& w2);
double osszeg_1(double x1, double x2);
double osszeg_2(double *x1, double *x2);
double osszeg_ref(double& w1, double& x2);
void kiir(double x, double y);
void main()
{
  double d1,d2,r1,r2;
  olvas(&d1,&d2);
  kiir(d1,d2);
  cout << "\nosszeg_1: " << osszeg_1(d1,d2);
  cout << "\nosszeg_2: " << osszeg_2(&d1,&d2) << endl;

  ref_olvas(r1,r2);
  kiir(r1,r2);
  cout << "\nosszeg_ref: " << osszeg_ref(r1,r2) << endl;

}
void olvas( double *x1, double *x2)
{
  cout << "\n1. adat: "; cin >> *x1;
  cout << "2. adat: "; cin >> *x2;
}
void ref_olvas(double& w1, double& w2)
{
  cout << "\n1. adat: "; cin >> w1;
  cout << "2. adat: "; cin >> w2;
}
double osszeg_1(double x1, double x2)
{
  return (x1+x2);
}
double osszeg_2(double *x1, double *x2)
{
  return (*x1+*x2);
}
```

```
double osszeg_ref(double& w1, double& w2)
{
  return(w1+w2);
}
void kiir(double x, double y)
{
  cout << "Beolvasott adatok : " << x << " " << y << endl;
}
```

A program futásának eredménye:

```
1. adat: 2
2. adat: 3
Beolvasott adatok : 2 3

osszeg_1: 5
osszeg_2: 5

1. adat: 5
2. adat: 6
Beolvasott adatok : 5 6

osszeg_ref: 11
```

A pointer típusú változók esetén használt hivatkozást az alábbi program mutatja be.

```
/* ref_2.cpp */
#include <iostream.h>
#include <stdlib.h>
void olvas( double *x1, double *x2);
void ref_olvas(double& w1, double& w2);
void kiir(double x, double y);
void main()
{
  double *d1,*d2;
  d1 = (double*)malloc(sizeof(double));
  d2 = (double*)malloc(sizeof(double));
  olvas(d1,d2);
  kiir(*d1,*d2);
  ref_olvas(*d1,*d2);
  kiir(*d1,*d2);
}
void olvas( double *x1, double *x2)
{
  cout << "1. adat: "; cin >> *x1;
  cout << "2. adat: "; cin >> *x2;
}
```

```
void ref_olvas(double& w1, double& w2)
{
  cout << "1. adat: "; cin >> w1;
  cout << "2. adat: "; cin >> w2;
}
void kiir(double x, double y)
{
  cout << "Beolvasott adatok : " << x << " " << y << endl;
}
```

A program futásának eredménye:

```
1. adat: 1
2. adat: 2
Beolvasott adatok : 1 2
1. adat: 3
2. adat: 4
Beolvasott adatok : 3 4
```

2.2. C++ operátorok

A C++ nyelv különféle operátorokkal bővítette a C nyelvet, azonban ezek nagy részének felhasználása az osztályokkal áll kapcsolatban.

::	érvényességikör operátor
new	dinamikus memóriafoglalás operátora
delete	dinamikus memóriafelszabadítás operátora
.*	osztálytagra történő indirekt hivatkozás operátora
->*	mutatóval megadott osztályobjektum tagjára való indirekt hivatkozás operátora

2.2.1. Az érvényességikör operátor ::

A C++ nyelvben az érvényességikör operátornak kétféle felhasználási lehetősége van. Az első értelmezésben a :: operátorral a program tetszőleges blokkjából hivatkozhatunk a globális (file) érvényességi körrel rendelkező nevekre.
A SCOPE.CPP program bemutatja a globális *n* változóra történő hivatkozást és módosítást.

```
/* scope.cpp */
#include <iostream.h>
void fuggv(int k);
int n = 5;          // globális változó
void main()
{
  int m;
  m = 2;
  cout << "main:  globális n = " << n << endl;
  fuggv(m);
  cout << "main:  fuggv hivás után globális n = " << n << endl;
}
void fuggv(int k)
{
 int n = 1;            //lokális n
 if ( k > n ) n = k;   //lokális n
 ::n++;                //globális n
 cout << "fuggv: lokális  n = " << n << endl;
 cout << "fuggv: globális n = " << ::n << endl;
}
```

A program futásának eredménye:

```
main:  globális n = 5
fuggv: lokális  n = 2
fuggv: globális n = 6
main:  fuggv hivás után globális n = 6
```

Az érvényességikör operátort használjuk, ha valamely osztály adattagjaira, illetve tagfüggvényeire hivatkozunk.

2.2.2. A new és a delete operátorok

C nyelven írt programok nagy része a szabad memóriát dinamikusan használja fel. A szükséges helyfoglalást az *stdlib.h* fejlécfájlban *malloc*, *calloc* stb., a felszabadítást a *free* könyvtári függvényekkel lehet elvégezni. A C++ nyelvben ezen könyvtári függvényeket helyettesíti a **new**, illetve a **delete** operátorok.

A **new** operátor a típusnak megfelelő méretű területet foglal le a szabad memóriában és a területre mutató pointert ad eredményül. A *NULL* pointer jelzi, ha a helyfoglalás nem sikerült.

```
/* new_p1.cpp */
#include <iostream.h>
#include <string.h>
void main()
{
   int    *i;
   long   *k;
   double *d;
   char   *c, *s;

   i = new int;
   k = new long;
   d = new double;
   c = new char;
   *i = 2; *k = 12456; *d = 3.672; *c = '*';
   cout << "\n i : " << *i << endl;
   cout << " k : " << *k << endl;
   cout << " d : " << *d << endl;
   cout << " c : " << *c << endl;
   cout << " s : " << s;
   delete i; delete k; delete d; delete c;
}
```

A program futásának eredménye:

```
 i : 2
 k : 12456
 d : 3.672
 c : *
 s : sztring
```

Egy- és kétdimenziós tömbök létrehozása **new** operátorral.

A következő mintafeladat bemutatja a dinamikus helyfoglalást. Először deklaráljuk a

```
      int    *vekt;
```

A **new** hívásával **int** típusból *db* darabot hozunk létre a később látható *olvas_int* függvényben:

```
      x = new int[db];
```

A vektor elemeire hivatkozhatunk *x[i]*.

A *del_int* függvényben pedig felszabadítjuk a *vekt* számára lefoglalt memóriaterületet:

```
      delete[] x;
```

Két dimenziós tömböt (mátrixot) kétcsillagos pointer deklarálásával is létrehozhatunk:

```
double **tomb_d;
```

Az *olvas_d* függvényben először lefoglaljuk a mátrix sorait megcímző pointer tömböt, majd ezeknek értéket adva, minden egyes sornak helyet foglalunk:

```
w = new double*[db];
for (int i = 0; i< db; i++)
   w[i] = new double[db];
```

A *del_d* függvényben történik a mátrix számára lefoglalt memóriaterület felszabadítása, szintén két menetben:

```
for( int i = 0; i<db; i++)
   delete[] w[i];
delete w;
```

Megjegyzés: vannak olyan C++ fordítók, amelyek a delete operátorral történő tömb felszabadításnál a tömbméret megadását is igénylik.

A mátrix elemeire kettős indexeléssel hivatkozhatunk:

```
w[i][j];
```

Megjegyezzük, hogy a C++ nyelvben utasítások közben is definiálhatunk változókat. Ez szokásos például a **for** ciklusok belsejében:

```
for( int i = 0; i<db; i++)
```

```
/* new_p2.cpp */
#include <iostream.h>
int* olvas_int(int db);
void kiir_int( int * x, int db);
double** olvas_d(int db);
void kiir_d( double **w, int db);
void del_int(int *x);
void del_d(double **w, int db);

void main()
{
   int    *vekt, n = 5, m = 3;
   double **tomb_d;
   vekt = olvas_int(n);
   kiir_int(vekt, n);
   tomb_d = olvas_d(m);
```

```
   kiir_d(tomb_d,m);
   del_int(vekt);
   del_d(tomb_d,m);
}

int* olvas_int(int db)
{
  int *x;
  x = new int[db];
  cout << "A vektor elemeinek feltöltése: \n";
  for(int i = 0; i<db; i++)
  {
    cout << "adat[" << i << "] = "; cin >> x[i];
  }
  return x;
}

void kiir_int( int *x, int db)
{
  cout << "\nA vektor elemei: ";
  for( int i = 0; i<db; i++)
  {
    cout << x[i] << "  ";
  }
  cout << endl;
}

void del_int(int *x)
{
    cout << "\nA vektor elemeinek felszabadítása";
    delete[] x;
}

double** olvas_d(int db)
{
 double **w;
   w = new double*[db];
   for (int i = 0; i< db; i++)
    w[i] = new double[db];
  cout << "\nA tömb elemeinek feltöltése\n";
  for(i = 0; i<db; i++)
   for(int j = 0; j<db; j++)
   {
     cout << "adat[" << i << "," << j << "] = ";
     cin >> w[i][j];
   }
  return w;
}

void kiir_d( double **w, int db)
{
  cout << "\nA tömb elemei: " << endl;
```

```
  for( int i = 0; i<db; i++)
  {
    for ( int j = 0; j<db ; j++)
    {
      cout << w[i][j] << "  ";
    }
    cout << endl;
  }
}
void del_d(double **w, int db)
{
  cout << "\nA tömb elemeinek felszabadítása";
  for( int i = 0; i<db; i++)
     delete[] w[i];
  delete w;
}
```

A program futásának eredménye:

```
A vektor elemeinek feltöltése:
adat[0] = 1
adat[1] = 2
adat[2] = 3
adat[3] = 4
adat[4] = 5
A vektor elemei: 1  2  3  4  5

A tömb elemeinek feltöltése
adat[0,0] = 3
adat[0,1] = 2
adat[0,2] = 1
adat[1,0] = 6
adat[1,1] = 5
adat[1,2] = 4
adat[2,0] = 9
adat[2,1] = 8
adat[2,2] = 7
A tömb elemei:
3  2  1
6  5  4
9  8  7

A vektor elemeinek felszabadítása
A tömb elemeinek felszabadítása
```

A C++ -ban is használhatjuk a ***malloc*** és a ***free*** függvényeket, de helyettük a **new** és a **delete** operátorok használata az ajánlott. Ugyanis ún. objektumokat csak ezekkel hozhatunk létre dinamikusan. Ha a **new** operátorral foglalunk helyet egy osztályobjektum számára, akkor meghívódik az osztály konstruktora,

illetve a **delete** művelet végrehajtásakor aktiválódik az osztály destruktora, míg a ***malloc*** ill. ***free*** függvények hívásakor nem történik meg.

Figyelem!

A **new - delete** illetve a **malloc - free** szigorúan párban használandó. A **delete** operátorral kizárólag csak new segítségével foglalt memóriát szabadíthatunk fel.

3. A C++ mint objektum-orientált nyelv

Ebben a fejezetben ismerkedünk meg az Objektum-Orientált Programozás (röviden OOP) lényegével, tulajdonságaival és lehetőségeivel.

3.1. Az OOP alapjai

Az OOP a természetes gondolkodást, cselekvést közelítő programozási mód, amely a programozási nyelvek tervezésének következetes fejlődése következtében alakult ki. Az így létrejött nyelv sokkal strukturáltabb, sokkal modulárisabb és absztraktabb, mint egy hagyományos programozási nyelv. Egy objektum-orientált nyelvet három fontos dolog jellemez. Ezek a következők:

- Az *egységbezárás* (encapsulation) azt jelenti, hogy az adatstruktúrákat és az adott struktúrájú adatokat kezelő függvényeket (a korábbi terminológiával élve metódusokat) kombináljuk; azokat egy egységként kezeljük, és elzárjuk őket a külvilág elől. Az így kapott egységeket *objektumoknak* nevezzük. Az objektumoknak megfelelő tárolási egységek típusa a C++-ban az *osztály* (**class**).
- Az *öröklés* (*inheritance*) azt jelenti, hogy adott, meglévő osztályokból származtatott újabb osztályok öröklik a definiálásukhoz használt alaposztályok már létező adatstruktúráit és függvényeit, ugyanakkor újabb tulajdonságokat is definiálhatnak, vagy régieket újraértelmezhetnek. Így egy osztályhierarchiához jutunk.
- A *sokalakúság (polymorphism)* alatt azt értjük, hogy egy adott tevékenység (*metódus*) azonosítója ugyanaz lehet egy adott osztályhierarchián belül, ugyanakkor a hierarchia minden egyes osztályában a tevékenységet végrehajtó függvény megvalósítása az adott osztályra nézve specifikus. Az ún. virtuális függvények lehetővé teszik, hogy egy adott metódus konkrét végrehajtási módja csak a program futása során derüljön ki. Ugyancsak a sokalakúság fogalomkörébe tartozik az ún. *overloading*, aminek egy sajátos esete a C nyelv szabványos operátorainak átdefiniálása, értelmezésük kiterjesztése, az ún. *operator overloading*.

A fenti a tulajdonságok együtt azt eredményezik, hogy programjaink kódja sokkal strukturáltabbá, könnyebben bővíthetővé, könnyebben karbantarthatóbbá válik, mint hagyományos, nem objektum-orientált technikával írnánk őket. Hogy a C++ előnyeit élvezhessük, kicsit módosítanunk kell a programozásról alkotott képünket. Ebben segít a fenti három tulajdonság részletesebb tárgyalása.

3.2. Egységbezárás

A C++ egyik fontos tulajdonsága, hogy lehetőségünk van az adataink és az adatokon műveleteket végző programkód összeforrasztására, egy egységbe zárására, egy osztályba foglalására. (Ez az ún. *encapsulation*.)

3.2.1. Alapfeladat: Műveletvégzés struct használatával

Alapfeladatként olyan programot mutatunk be, amely a négy alapműveletet hajt végre. Ennek a feladatnak a különféle módon történő megoldása összehasonlításként szolgál a C és a C++ nyelv közötti különbségek megértéséhez, valamint segíti a hagyományos programozásról az objektum-orientált programozásra való áttérést.

Írjunk olyan műveletvégző programot, amely a négy alapműveletet hajtja végre! A feladat megoldásához használjunk struktúrát.
A feladat megoldását a MUV_C2.C állomány tartalmazza.

A *Muveletvegzes* struktúra leírja a működéséhez szükséges adatokat: az operandusokat a **float** típusú *x,y* adattagok, a műveleti jelet a **char** típusú *muv* adattag és a művelet eredményét az **float** típusú *eredmeny* adattag tárolja.

```
typedef struct Muveletvegzes{
                                float x,y;
                                float eredmeny;
                                char  muv;
                               } MUVELET;
```

Írjunk egy *Olvas* függvényt, amely egy paraméteren keresztül adja vissza az adatokkal feltöltött struktúrát.

```
      void Olvas(MUVELET *sz)
```

Írjunk egy *Szamol* függvényt, amely a *MŰVELET* típusú struktúra adatai alapján elvégzi a műveletet, és a művelet eredményét nem csak a **return** utasításban adja vissza, hanem a *MUVELET* struktúra *eredmeny* mezejét is kitölti.

A fenti feladatokat megvalósító a *Szamol* függvénynek az alábbi prototípussal kell rendelkeznie:

```
float Szamol(MUVELET *sz);
```

Látható, hogy a paraméterlistán átvesszük a *MUVELET* típusú struktúrára mutató pointert és a művelet elvégzése után visszaadjuk az eredményt, amely **float** típusú (MUV_C2.C)

```
#include <stdio.h>
typedef struct Muveletvegzes{
                                float x,y;
                                float eredmeny;
                                char  muv;
                               } MUVELET;
void Olvas(MUVELET *sz)
{
 float x;
 printf("művelet (+,-,*,/): "); scanf("%c",&sz->muv);
 printf("1. adat: "); scanf("%f",&x); sz->x = x;
 printf("2. adat: "); scanf("%f",&x); sz->y = x;
}
float Szamol(MUVELET *sz)
{
 sz->eredmeny = 0;
 switch (sz->muv)
 {
   case '+' : sz->eredmeny = sz->x+sz->y; break;
   case '-' : sz->eredmeny = sz->x-sz->y; break;
   case '*' : sz->eredmeny = sz->x*sz->y; break;
   case '/' : sz->eredmeny = sz->x/sz->y; break;
 }
 return sz->eredmeny;
}
void main()   /* művelet végzése */
{
 MUVELET z;
   Olvas(&z);
   printf("%6.2f %c %6.2f = %6.2f\n",z.x,z.muv,z.y,Szamol(&z));
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 4
2. adat: 3
  4.00 *    3.00 =  12.00
```

Készítsük el az előző feladat megoldásának objektum-orientált változatait! Ha ezen feladatok különböző megoldásait végig követjük, akkor már világosan látni fogjuk a legalapvetőbb különbségeket a C és a C++ nyelv között, és megismerhetjük a leglényegesebb objektum-orientált alapelveket.

3.2.2. A Műveletvégzés feladat objektum-orientált változatai

A hagyományos C-ben az a szokásos megoldás, hogy az adatstruktúráinkat és a hozzájuk tartozó függvényeket egy önálló, külön fordítható forrásmodulban helyezzük el. Ez a megoldás már elég elegáns, de az adatok és az őket kezelő függvények között még nincs egyértelmű összerendeltség, továbbá más programozók egy másik modulból közvetlenül is hozzáférhetnek az adatainkhoz, anélkül, hogy az adatok kezelésére szolgáló függvényeinket használnánk. Ilyen esetekben az alap-adatstuktúra megváltozása fatális hibát okozhat egy nagyobb csoportmunkában.

A C++-ban speciális egységfajták, az osztályok szolgálnak arra, hogy az adatokat és a hozzájuk rendelt függvényeket egy egységként, egy objektumként kezeljük. Az osztályok alaptípusa a C++-ban a **class**, amely hasonlít a hagyományos C-ből megismert **struct**-ra és a **union**-ra. A C++-ban a **struct** és a **union** is egy bizonyos fajta osztály. A **union**-ra a hagyományos értelemben továbbra is szükség van (különböző típusú adatok tárolására azonos memóriaterületen, bit-mezők kezelése), a **struct** kulcsszót pedig a C++-ban alapvetően a hagyományos C-vel való kompatibilitás biztosítása végett tartották meg.

A C++-ban a **class** deklaráció a C **struct** kiterjesztése, mindkét struktúra tartalmazhat adatmezőket (adattag - *data members*), azonban a C++-ban ezen adatmezőkhöz különféle műveleteket, ún. tagfüggvényeket (*member function*) is megadhatunk. A C++-ban egy osztály típusú tárolási egység függvényeket kombinál adatokkal, és az így létrejött kombinációt elrejtjük, elzárjuk a külvilág elől. Ezt értjük az egységbezárás alatt. Egy **class** deklaráció hasonló a jól ismert struktúradeklarációhoz:

```
class osztálynév
{
        adatmezők
  public:
        tagfüggvények
};
```

A hagyományos C struktúrák (**struct**) és a C++ osztályok (**class**) között a fő különbség a mezőkhöz való hozzáférésben van. A C-ben egy struktúra mezői (a megfelelő érvényességi tartományon belül) szabadon elérhetőek, míg a C++-ban egy **struct** vagy **class** minden egyes mezőjéhez való hozzáférés önállóan kézbentartható azáltal, hogy publikusnak, privátnak vagy védettnek deklaráljuk a **public**, **private** és **protected** kulcsszavakkal. (A hozzáférési szinteket később részletesebben is tárgyaljuk.) A **class** kulcsszóval definiált osztályban az adatmezők alapértelmezés szerint **private** hozzáférésűek, bár ez is módosítható. Az OO-re jellemző, hogy az adatmezőket privátnak, a tagfüggvényeket pedig publikusnak deklaráljuk. Fontos megjegyezni, hogy egy C++ osztály önállóan is típus-értékű, nem kell **typedef** segítségével típusazonosítót hozzárendelni.

Ebben a feladatsorban a már struktúrával megoldott, egyszerű műveletvégző feladatot használjuk fel az objektum-orientált programtervezés és programírás főbb lépéseinek bemutatására. A feladatok során az egyszerű, nem objektum-orientált programhoz némileg hasonló megoldástól jutunk el a bonyolultabb, az objektum-orientált programozás lényegét és előnyeit bemutató megoldásokig.

3.2.2.1. Statikus helyfoglalású objektumpéldány

Tervezzünk olyan objektumot, amely alkalmas arra, hogy műveletvégzéshez szükséges adatokat tároljon és elvégezze a négy alapművelet, valamint az eredményt a művelet jelölésével kiírja! A feladat megoldásához használjunk statikus helyfoglalású objektumpéldányt!
A feladat megoldását a MUV_CPP2.CPP állomány tartalmazza.

Az alábbi adattagok és tagfüggvények szükségesek a feladat megoldásához:

Adattag	***Típusa***	***Tartalma***
x,y	**float**	adatok a műveletvégzéshez,
muv	**char**	műveleti jel,
eredmeny	**float**	a művelet eredménye
Tagfüggvény	***Típusa***	***Feladata***
Init	**void**	adattagok inicializálása
Szamol	**void**	a kívánt művelet elvégzése.
Kiir	**void**	az eredmény kiírása.

Definiáljunk egy *Muvelet* objektumtípust, amely a fentiekben leírt adattagokkal és tagfüggvényekkel rendelkezik:

```
class Muvelet
{
  public:
       float x,y;
       float eredmeny;
       char  muv;
  public:
       void Init(char muv1, float x1, float y1);
       void Szamol();
       void Kiir();
 };
```

Az osztálydefiníció két részből áll:

- Az osztály feje a **class** alapszó után az osztály nevét tartalmazza. A másik rész az osztály törzse, amelyet kapcsos zárójelek fognak közre és pontosvessző vagy objektumlista zár.
- Az osztálydefiníció az adattagokon és tagfüggvényeken kívül a tagokhoz való hozzáférést szabályozó **public**, **private** és **protected** kulcsszavakat is tartalmazhatja.
  - A **public** tag bárhonnan elérhető a programon belül, ahonnan maga az objektum elérhető. Az adatrejtés elvének érvényesüléséhez ajánlott, hogy **public** eléréssel csak tagfüggvényeket deklaráljunk. A **public** kulcsszó után kettőspont következik és az adattagokat a struktúra adatmezőihez hasonlóan adjuk meg.
  - A **protected** tagok külső függvények számára **private**, de a származtatott osztályok tagfüggvényei számára **public** elérésűek. Az osztálytagok **protected** elérése az osztályhierarchia kialakításánál, vagyis az öröklődésnél (*inheritance*) játszik szerepet.
  - A **private** tagokat csak az osztály saját tagfüggvényeiből, illetve az osztály "barátaiból" (**friend**) érhetjük el. A külső függvények és a származtatott osztályok tagfüggvényei (habár a **private** tagok is öröklődnek), nem rendelkeznek hozzáférési joggal a **private** osztálytagokhoz. Általában az osztályok adattagjait **private** vagy **protected** eléréssel deklarálják.

Megjegyezzük, hogy

- egy osztály tagfüggvényei az alapértelmezés szerint a kérdéses osztály saját adattagjain végeznek műveletet, így az adattagokat nem kell a argumentumként átadni.
- egy osztály teljes definiálásához a tagfüggvények definícióját is meg kell adnunk. A tagfüggvény definícióban az osztály nevét két kettősponttal elválasztva kapcsoljuk a tagfüggvény nevéhez, felhasználva a érvényességi kör (::) operátort. Erre azért van szükség, mivel a különböző osztályoknak lehetnek azonos nevű tagfüggvényei.
- a *Muvelet* osztály *Init* tagfüggvényének feladata az osztály adattagjainak inicializálása. A tagfüggvény a kezdőértékeket veheti a paraméterlistáról, vagy konstansokból, vagy beolvashatja billentyűzetről.
- a *Muvelet* osztály *Szamol* tagfüggvénye teljesen azonosan számítja ki a műveleteket, mint ahogy azt a nem objektum-orientált változat teszi. A nem objektum-orientált változatnál természetesen a paraméterként megadott struktúra adatmezőivel végzünk műveleteket. Az objektum-orientált változatnál szembetűnő különbség, hogy a *Szamol* tagfüggvénynek nincs paraméterlistája, mert az *Init* tagfüggvény az adattagoknak már átadta azt a kezdeti értéket, amivel számolnia kell és az eredmény is adattagban keletkezik.

A következő változóknak foglalunk helyet a ***main*** függvényben:

A változó neve	***Típusa***	***Tartalma***
x1,y1	**float**	két adat beolvasásával a műveletek operandusait tárolja,
muv1	**char**	a műveleti jel,
z	*Muvelet*	statikus helyfoglalású objektumpéldány.

```
/* muv_c2.cpp */
include <iostream.h>
class Muvelet
{
 public:
        float x,y;
        float eredmeny;
        char  muv;
 public:
        void Init(char muv1, float x1, float y1);
        void Szamol();
        void Kiir();
};
```

```
void Muvelet::Init(char muv1, float x1, float y1)
{
  muv = muv1;
  x = x1;
  y = y1;
  eredmeny = 0;
}
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y
       << " = " << eredmeny << endl;
}
void main()
{
 Muvelet z;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  z.Init(muv1,x1,y1);
  z.Szamol();
  z.Kiir();
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): +
1. adat: 2
2. adat: 3
2 + 3 = 5
```

Mivel a *z* statikus objektum, a definícióval már megtörtént számára a helyfoglalás a memóriában, ezért meghívhatjuk az *Init* tagfüggvényt a beolvasott *muv1,x1,y1* változókat átadva argumentumként, és ezzel megtörténik az objektum inicializálása.

Az objektumpéldány adattagjainak és tagfüggvényeinek elérése kétféleképpen történik:

– statikus objektumok esetén a struktúránál megismert pont (.) operátorral,

- míg a nyíl (->) operátorral dinamikusan létrehozott, pointer segítségével elérhető objektumok esetén

A példákból jól látható, az objektum-orientált nyelvekben az adatok és az adatokon végzett műveletek egyenrangúak és zárt egységet alkotnak, ellentétben a hagyományos programozási nyelvekkel.

3.2.2.2. Tagfüggvények inline definíciója

A tagfüggvényt *inline* (sorok között) is megadhatjuk, ha az osztálydefinícióban a tagfüggvény prototípusa helyett annak törzsét is elhelyezzük.

Egy *inline* függvény elég kis terjedelmű ahhoz, hogy helyben, a függvényhívás helyére behelyettesítve le lehessen fordítani. Ebben a tekintetben egy *inline* függvény olyan, mintha makró lenne, azaz a függvényhívásokkal járó adminisztrációra nincs szükség. Rövid, egy-két soros függvények esetében ez sokkal hatékonyabb, sokkal olvashatóbb megoldást jelent, mint a makró-definíció. Az ún. explicit **inline** függvények deklarálására szolgál a C++ **inline** kulcsszava.

Az *inline* függvények a hagyományos C előfeldolgozó **#define** direktívájával definiálható makrókhoz hasonlítanak. Annyival fejlettebbek a makróknál, hogy egy ilyen függvény meghívása nem pusztán szöveghelyettesítés, hanem a definíció szerint generált kód másolódik be a hívások helyére, így a makrókkal kapcsolatban említett mellékhatások jelentkezésének is kisebb a veszélye.

A feladat megoldását, amelyben az *Init* tagfüggvényt *inline* módon adtuk meg, a MUV_INL.CPP állomány tartalmazza:

```
class Muvelet
{
  private:
      float x,y;
      float eredmeny;
      char  muv;
  public:
      void Init(char muv1, float x1, float y1)  // inline megadási mód
      {
        muv = muv1;
        x = x1;
        y = y1;
        eredmeny = 0;
      }
      void Szamol();
      void Kiir();
};
```

A *Szamol* tagfüggvényt nem definiálhatjuk *inline* módon. Ha megkísérelnénk, az alábbi hibajelzést kapnánk: `Containing switch are not expended inline` (**Switch** utasítást tartalmazó függvény nem lehet *inline* módon megadni.)

3.2.2.3. Statikus helyfoglalású objektumpéldányok

Módosítsuk a MUV_CPP2.CPP programot úgy, hogy két különböző műveletvégző adatait olvassuk be és számítsuk ki a műveleteket!
A feladat megoldását az MUV_ST2.CPP állomány tartalmazza.

A statikus helyfoglalású objektumpéldányokat a program adatterületén hozza létre a fordító, több példány deklarálása esetén csak az adattagokat többszörözi, így minden egyes példány saját adatterülettel rendelkezik. Az objektum tagfüggvényei azonban csak egyetlen példányban kerülnek a kódterületre és az objektumpéldányok közösen használják a tagfüggvényeket

Minden tagfüggvény, még a paraméter nélküliek is ((**void**)) rendelkeznek egy nem látható (*implicit*) paraméterrel: ez a **this**, amelyben a hívás során az aktuális objektumpéldányra mutató pointert ad át a C++ és minden adattag-hivatkozás automatikusan

```
this->adattag
```

kifejezésként kerül be a kódba.

A feladat megoldásához csak a definíciót és a ***main*** függvényt kell módosítanunk. Definiálunk két statikus helyfoglalású *z1* és *z2* objektumpéldányt:

```
void main()
{
 Muvelet z1,z2;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  z1.Init(muv1,x1,y1);
  z1.Szamol();
  z1.Kiir();
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  z2.Init(muv1,x1,y1);
  z2.Szamol();
  z2.Kiir();
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): +
1. adat : 3
2. adat : 4
3 + 4 = 7

művelet (+,-,*,/): *
1. adat : 4
2. adat : 6
4 * 6 = 24
```

3.2.2.4. Az adattagok privát (private) elérése

Változtassuk meg *Muvelet* osztály adattagjainak **public** elérését **private** elérésre! Ezt követően az alábbi műveletre

```
cout << "Eredmeny: " << z.eredmeny;
```

hibajelzést kapunk:

```
'Muvelet::eredmeny' is not accessible
```

vagyis a *Muvelet* osztály *eredmeny* adattagja közvetlenül nem érhető el. Bővítsük a *Muvelet* osztály definícióját új tagfüggvénnyel, amellyel már elérhetjük az *eredmeny* **private** adattagot:

```
float Eredmeny();
```

A feladat megoldását a MUV_PRI.CPP program tartalmazza:

```
#include <iostream.h>

class Muvelet
{
      private:
             float x,y;
             float eredmeny;
             char  muv;

      public:
             void Init(char muv1, float x1, float y1);
             float Eredmeny();
             void Szamol();
             void Kiir();
```

```
void Muvelet::Init(char muv1, float x1, float y1)
{
        muv = muv1;
        x = x1;
        y = y1;
        eredmeny = 0;
}
float Muvelet::Eredmeny()
{
        return eredmeny;
}
void Muvelet::Szamol()
{
 switch (muv)
 {
        case '+' : eredmeny = x+y; break;
        case '-' : eredmeny = x-y; break;
        case '*' : eredmeny = x*y; break;
        case '/' : eredmeny = x/y; break;
 }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y
        << " = " << eredmeny << endl;
}
void main()
{
 Muvelet z;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  z.Init(muv1,x1,y1);
  z.Szamol();
  z.Kiir();
  cout << "Eredmeny: " << z.Eredmeny();
}
```

3.2.2.5. Az adattagok protected elérése

Ha az adattagok **public** elérését **protected**-re változtatjuk, akkor ebben az esetben is tagfüggvényeket kell használnunk az adattagok értékének kiolvasására.

3.2.2.6. A new és a delete operátorok használata

Ahhoz, hogy a szabad memóriaterületet dinamikusan kezeljük a programból, mutatókat kell használunk. A C programban könyvtári függvényekkel végezzük el a mutató számára a memóriafoglalást (***malloc***()), illetve a felszabadítást (***free***()).

A C++ nyelvben a **new** és a **delete** operátorok nyelvdefiníció szintjén helyettesítik a fenti könyvtári függvényeket:

– a **new** operátor az operandusában megadott típusnak megfelelő méretű területet foglal le a szabad memóriából és a memóriaterületre mutató pointert ad eredményül.

  Általános formája:

  <::> **new** *<elhelyezés > típus név <(kezdőérték)>*

  <::> **new** *<elhelyezés >* (*típus név*) *<(kezdőérték)>*

  :: operátorral a **new** globális verzióját használhatjuk

  elhelyezés a **new** átdefiniált verziójánál használjuk

  kezdőérték megadásával a lefoglalt területre kezdőértéket adhatunk. Ez a tömbök esetén nem használható.

– a **delete** operátor a **new** operátor által lefoglalt területet felszabadítja.

  Általános formája:

  <::> **delete** *<típusátalakító kifejezés>*

  <::> **delete [ ]** *<típusátalakító kifejezés>*

  delete *<tömbazonosító>* [];

Példaként hozzunk létre egy 10 elemű dupla pontosságú tömböt dinamikusan:

```
double *x;
x = new double[10];
```

Szabadítsuk fel a tömb számára lefoglalt memóriaterületet:

```
delete[] x;
```

A **new** és a **delete** operátorokat átdefiniálhatjuk mutatók nyilvántartására. A **new** operátor használata esetén eltároljuk a megszületett mutatókat és a **delete** operátornál pedig adminisztráljuk a mutató megszüntetését. A program futása végén listát készíthetünk a fel nem szabadított mutatókról. A program akkor működik helyesen, ha nincs felszabadítatlan mutató.

A **new** operátor átdefiniálása:

```
void * operator new(size_t Type_size); // nem tömb típus esetén
void * operator new[](size_t Type_size);// tömb típus esetén
```

A **delete** operátor átdefiniálása

Nem tömbtípus esetén:

```
void operator delete (void *Type_ptr, [size_t Type_size]);
```

Tömbtípus esetén:

```
void operator delete [](void *Type_ptr, [size_t Type_size]);
```

3.2.2.7. A konstruktor

A C++ programokban az objektumok inicializálását speciális tagfüggvények, a konstruktorok végzik.

A konstruktor tulajdonságai és feladatai:

- A konstruktor olyan speciális tagfüggvény, mely nevének meg kell egyeznie az őt tartalmazó osztály nevével.
- A konstruktor nem rendelkezik visszatérési értékkel, de ugyanúgy viselkedik, mint bármely más tagfüggvény.
- A konstruktor nem foglal memóriát a létrejövő objektum számára – ezt a fordító végzi az alaptípusoknál használt módszerrel.
- Ha az objektum mutatót tartalmaz, akkor a konstruktorban gondoskodnunk kell mutató által kijelölt terület létrehozásáról.
- A konstruktor feladata a már lefoglalt memóriaterület inicializálása.
- A konstruktor át is definiálható, így adott a lehetőség a többféle inicializálás megvalósítására. Egy osztálynak ezért többfajta konstruktora lehet, például nincs paramétere, mert konstans értékeket ad az adattagoknak, vannak paraméterei, illetve csak néhány paramétere van. Ilyen esetben mindig az a

konstruktor szólal meg, amelyeknek a prototípusa megegyezik a hívott konstruktor prototípusával.

- Egy speciális konstruktor az ún. *másoló konstruktor* (*copy constructor*). Ha S egy osztály, akkor a hozzá tartozó másoló konstruktoralakja

```
S::S(S&)
```

vagy

```
S::S(const S&)
```

Másoló konstruktorok működnek, amikor a deficícióban egy másik objektummal inicializálunk:

```
S s1;
S s2 = s1;
S s3(s1);
```

A fent felsorolt tulajdonságok alapján kijelenthetjük, hogy a konstruktor több mint egy *Init* tagfüggvény.

3.2.2.8. A destruktor

A **new** operátor által az objektum számára lefoglalt memória mindaddig lefoglalt marad, míg fel nem szabadítjuk. A C++ nyelvben az ily módon lefoglalt területek felszabadításáról egy speciális tagfüggvényben, a destruktorban kell gondoskodnunk. A destruktorok dolga tehát egy objektumpéldány megszüntetésével kapcsolatos kiegészítő műveletek (pl. memória felszabadítása, fájlok lezárása, stb.) elvégzése.

- A destruktor nevét a hullám karakterrel (~) egybeírt osztálynévként kell megadni. A destruktor hasonlóan a konstruktorhoz, szintén nem rendelkezik visszatérési típussal.
- Ha az osztály rendelkezik destruktorral, a fordító minden olyan esetben meghívja azt, amikor egy objektumpéldány érvényessége megszűnik, (pl. lokális változók megszűnése függvényből való visszatéréskor, dinamikus objektumpéldány felszámolása **delete** operátorral).

  A **new** operátorral létrehozott objektumok esetén a destruktor aktivizálása csak a **delete** operátor aktivizálásával lehetséges.
- Egy destruktor lehet virtuális (**virtual**) függvény, azonban a konstruktorok nem lehetnek virtuálisak.
- Ha nem definiálunk mi magunk destruktort egy adott osztályhoz, a C++ fordító a saját alapértelmezése szerinti változatát használja.

3.2.2.9. Konstruktor használata statikus objektum esetén

A *Muvelet* osztály tagfüggvényeként definiáljunk konstruktort:

```
class Muvelet
{
      private:
            float x,y;
            float eredmeny;
            char  muv;
      public:
            Muvelet(char muv1, float x1, float y1);
            void Szamol();
            void Kiir();
};
```

A konstruktort ugyanúgy deklaráljuk, mint minden egyéb tagfüggvényt. Figyeljük ugyanakkor meg, hogy a konstruktor neve és az osztály neve ugyanaz (*Muvelet*). Innen tudja a fordítóprogram, hogy az adott tagfüggvény egy konstruktor. Figyeljük meg azt is, hogy mint minden függvénynek, a konstruktornak is lehetnek paraméterei, a konstruktor törzse pedig egy szabályos függvénytörzs. Ez azt jelenti, hogy egy konstruktor hívhat más függvényeket és minden, a hatáskörébe tartozó adathoz hozzáférhet. Az egyetlen, de lényeges különbség, hogy egy konstruktornak nem lehet típusa, még **void** sem! (Így értéket nem adhat vissza,)

Statikus objektum esetében a konstruktort már a definíciónál hívni kell az alábbi módon:

```
Muvelet z(muv1,x1,y1);// inicializálás
```

Ez a programsor aktivizálja a fent definiált konstruktort, és ennek következtében a *muv*, *x* és *y* rendre átveszi a *muv1*, *x1* és *y1* változók tartalmát. Az overloading a konstruktorok esetében is alkalmazható, így az osztálynak több konstruktora is lehet, és mindig az argumentumlista alapján dől el, hogy melyik változtot kell aktivizálni. Ha mi magunk nem definiálunk konstruktort, akkor a C++ egyet létrehoz, amelynek nincsenek paraméterei.

Az osztályunk nem tartalmaz dinamikus helyfoglalású adattagokat, ezért nem írtunk hozzá destruktort (MUV_CPP3.CPP).

```
#include <iostream.h>
class Muvelet
{
    public:
           float x,y;
           float eredmeny;
           char  muv;
    public:
           Muvelet(char muv1, float x1, float y1);
           void Szamol();
           void Kiir();
};
Muvelet::Muvelet(char muv1, float x1, float y1)
{
    muv = muv1;
    x = x1;
    y = y1;
    eredmeny = 0;
}
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y
       << " = " << eredmeny << endl;
}
void main()
{
 char muv1;
 float x1,y1;
   cout << "művelet (+,-,*,/): "; cin >> muv1;
   cout << "1. adat: "; cin >> x1;
   cout << "2. adat: "; cin >> y1;
   Muvelet z(muv1,x1,y1);
   z.Szamol();
   z.Kiir();
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): /
1. adat: 6
2. adat: 2
6 / 2 = 3
```

3.2.2.10. Paraméterezett típusok (templates)

A sablonok (*template*) használata lehetővé teszi, hogy egymással logikai kapcsolatban álló függvények és osztályok családját hozzuk létre.

Függvénysablonok

Ha egy olyan függvényt akarunk készíteni, amely különféle típusú paraméterekkel működjön, használnunk kell az átdefiniálás (*overloading*) mechanizmusát. Az átdefiniálás során a függvények törzse gyakorlatilag nem változik. A C++-ban típus sablonnal (template-kel) paraméterezett függvényeket és különböző adattípusokat visszaadó függvényeket is használhatunk.

Írjunk olyan adatcsere függvényeket, amely felcseréli két egész, valós és karakter típusú változó tartalmát!

```
/* csere.cpp */
#include <iostream.h>
void adatcsere(int& v1, int& v2);
void adatcsere(float& v1, float& v2);
void adatcsere(char& v1, char& v2);
void main()
{
  int   i1 = 10, i2 = -4;
  float f1 = 1.5, f2 = 2.8;
  char  c1 = '*', c2 = '#';
  cout << "csere előtt  i1 : " << i1 << " i2 : " << i2 << endl;
  adatcsere(i1,i2);
  cout << "csere után   i1 : " << i1 << " i2 : " << i2 << endl;
  cout << "csere előtt  f1 : " << f1 << " f2 : " << f2 << endl;
  adatcsere(f1,f2);
  cout << "csere után   f1 : " << f1 << " f2 : " << f2 << endl;
  cout << "csere előtt  c1 : " << c1 << " c2 : " << c2 << endl;
  adatcsere(c1,c2);
  cout << "csere után   c1 : " << c1 << " c2 : " << c2 << endl;
}
void adatcsere(int& v1, int& v2)
{
  int t;
  t  = v1;
  v1 = v2;
  v2 = t;
}
```

```
void adatcsere(float& v1, float& v2)
{
  float t;
  t  = v1;
  v1 = v2;
  v2 = t;
}
void adatcsere(char& v1, char& v2)
{
  char t;
  t  = v1;
  v1 = v2;
  v2 = t;
}
```

A program futásának eredménye:

```
csere előtt  i1 : 10 i2 : -4
csere után   i1 : -4 i2 : 10
csere előtt  f1 : 1.5 f2 : 2.8
csere után   f1 : 2.8 f2 : 1.5
csere előtt  c1 : * c2 : #
csere után   c1 : # c2 : *
```

Az *adatcsere* függvény paramétereinek sablonnal történő megoldása.

```
/* csere.cpp */
#include <iostream.h>
template<class T>
void adatcsere(T& v1, T& v2);
void main()
{
  int   i1 = 10, i2 = -4;
  float f1 = 1.5, f2 = 2.8;
  char  c1 = '*', c2 = '#';
  cout << "Template függvény paraméter\n";
  cout << "csere előtt  i1 : " << i1 << " i2 : " << i2 << endl;
  adatcsere(i1,i2);
  cout << "csere után   i1 : " << i1 << " i2 : " << i2 << endl;
  cout << "csere előtt  f1 : " << f1 << " f2 : " << f2 << endl;
  adatcsere(f1,f2);
  cout << "csere után   f1 : " << f1 << " f2 : " << f2 << endl;
  cout << "csere előtt  c1 : " << c1 << " c2 : " << c2 << endl;
  adatcsere(c1,c2);
  cout << "csere után   c1 : " << c1 << " c2 : " << c2 << endl;
}
```

```
template<class T>
void adatcsere(T& v1, T& v2)
{
  T t;
  t  = v1;
  v1 = v2;
  v2 = t;
}
```

A program futásának eredménye:

```
csere előtt  i1 : 10 i2 : -4
csere után   i1 : -4 i2 : 10
csere előtt  f1 : 1.5 f2 : 2.8
csere után   f1 : 2.8 f2 : 1
csere előtt  c1 : * c2 : #
csere után   c1 : # c2 : *
```

Írjunk olyan *minimum* függvényeket, amelyek két egész, valós és dupla pontosságú változó közül kiválasztja a kisebbet és annak értékét adja vissza!

```
/* min_pr.cpp */
#include <iostream.h>
int minimum( int , int );
float minimun(float , float );
double minimum(double , double );
void main()
{
 int i1 = 20, i2 = 3;
 float f1 = 3.56, f2 = -10.3;
 double d1 = -3.5, d2 = 12.6;

  cout << i1 << " " << i2 << " -> minimum: "
       << minimum(i1,i2) << endl;
  cout << f1 << " " << f2 << " -> minimum: "
       << minimum(f1,f2) << endl;
  cout << d1 << " " << d2 << " -> minimum: "
       << minimum(d1,d2) << endl;
}
int minimum( int v1, int v2)
{
  if (v1 < v2) return (v1); else return(v2);
}
float minimun(float v1, float v2)
{
   if (v1 < v2) return (v1); else return(v2);
}
double minimum(double v1, double v2)
{
   if (v1 < v2) return (v1); else return(v2);
}
```

A program futásának eredménye:

```
20 3 -> minimum: 3
3.56 -10.3 -> minimum: -10.3
-3.5 12.6 -> minimum: -3.5
```

A *minimum* megoldása függvénysablon használatával, amikor a visszatérési értéket is sablonnal definiáljuk:

```
/* min_prt.cpp */
#include <iostream.h>
template<class T> T minimum( T v1,T v2);
void main()
{
 int i1 = 20, i2 = 3;
 float f1 = 3.56, f2 = -10.3;
 double d1 = -3.5, d2 = 12.6;

  cout << i1 << " " << i2 << " -> minimum: "
       << minimum(i1,i2) << endl;
  cout << f1 << " " << f2 << " -> minimum: "
       << minimum(f1,f2) << endl;
  cout << d1 << " " << d2 << " -> minimum: "
       << minimum(d1,d2) << endl;
}
template <class T> T minimum( T v1, T v2)
{
  if (v1 < v2) return (v1); else return(v2);
}
```

A program futásának eredménye:

```
20 3 -> minimum: 3
3.56 -10.3 -> minimum: -10.3
-3.5 12.6 -> minimum: -3.5
```

Osztálysablonok

A típussablonnal paraméterezett osztály (***generic class***), lehetővé teszi, hogy más osztályok definiálásához a paraméterezett osztályt felhasználjuk, ezáltal egy adott osztálydefiníció minden típus esetén alkalmazható lesz.

Az osztálysablon használata statikus objektum esetén

Az osztálysablon segítségével az adattagok, illetve a tagfüggvények **int, float** (stb.) típusúként is viselkedhetnek, ezt a létrehozatalnál a fordító dönti el. Például:

```
template <class R>
```

```
class Muvelet
{
  public:
       R x,y;
       R eredmeny;
       char  muv;
  public:
       Muvelet(char muv1, R x1, R y1);
       void Szamol();
       void Kiir();
 };
```

A művelet **float** adatokat tud feldolgozni és a számítás eredménye is **float** lesz:

```
       Muvelet <float> pf(muv1,x1,y1);
       cout.setf(iostream::fixed, iostream::floatfield);
```

A művelet **int** adatokat tud feldolgozni és a számítás eredménye is **int** lesz:

```
      Muvelet <int> pi(muv1,x1,y1);
```

A feladat megoldását a MUV_TEMP.CPP tartalmazza.

```
#include <iostream.h>

template <class R>
class Muvelet
{
  public:
       R x,y;
       R eredmeny;
       char  muv;
  public:
       Muvelet(char muv1, R x1, R y1);
       void Szamol();
       void Kiir();
 };

template <class R>
Muvelet<R>::Muvelet(char muv1, R x1, R y1)
{
  muv = muv1;
  x = x1;
  y = y1;
  eredmeny = 0;
}
```

```
template <class R>
void Muvelet<R>::Szamol()
{
 switch (muv)
  {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
  }
}
template <class R>
void Muvelet<R>::Kiir()
{
   cout << x << " " << muv << " " << y << " = "
         << eredmeny << endl;
}

void main()
{
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  Muvelet <float> pf(muv1,x1,y1);
  cout.setf(iostream::fixed, iostream::floatfield);
  pf.Szamol();
  pf.Kiir();
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  Muvelet <int> pi(muv1,x1,y1);
  pi.Szamol();
  pi.Kiir();
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 3.1
2. adat: 2.3
3.1 * 2.3 = 7.13
művelet (+,-,*,/): *
1. adat: 3
2. adat: 2
3 * 2 = 6
```

3.2.2.11. Dinamikus helyfoglalású objektumpéldány

Az előző példáinkban csak statikus objektumokkal volt dolgunk (amelyek számára a fordítóprogram foglal le tárterületet), jóllehet a hagyományos C-ben már megszokhattuk, hogy változóink egy része számára mi magunk foglalunk le tártetületet, és a változókat megszüntetve felszabadítjuk a memóriát, ha az adott változókra többé már nincs szükségünk.

Természetesen a C++-ban is használhatunk dinamikus tárkezelést az objektumoknál. A dinamikus objektumok létrehozására a C++-ban a **new** operátor szolgál, amely aktiválja a konstruktort.

Használjunk dinamikus helyfoglalású objektumpéldányt a műveletek elvégzésére!
A feladat megoldását a MUV_CPP4.CPP állomány tartalmazza.

A *Muvelet* osztály konstruktora, amely a dinamikus objektumpéldány adattagjainak kezdőértéket ad:

```
Muvelet::Muvelet(char muv1, float x1, float y1)
{
  muv = muv1;
  x = x1;
  y = y1;
  eredmeny = 0;
}
```

A *Muvelet* destruktorát jelenleg üres törzzsel definiáltuk, mivel nincs olyan adattag, melyet az objektumban meg kellene szüntetni.

```
Muvelet::~Muvelet()
{ }
```

Az objektumpéldány dinamikus létrehozására mutatót használunk:

```
Muvelet *pz;
```

Ha az objektum dinamikus, akkor a **new** operátorral foglalunk memóriahelyet és megadjuk a konstruktornak átadandó paramétereket:

```
pz = new Muvelet(muv1,x1,y1);
```

A *Szamol* tagfüggvény meghívása a -> operátorral történik

```
pz->Szamol();
```

Ha **new**-val dinamikusan hozunk létre objektumokat, akkor a mi felelősségünk, hogy meg is szüntessük azokat, amikor már nincs rájuk szükségünk. Erre szolgál a **delete** operátor. Egy nem NULL értékű, tetszőleges objektumra mutató pointerre alkalmazva a **delete** operátort, aktivizálódik az adott típusú objektumhoz tartozó destruktor és a pointer által megcímzett tárterület felszabadul. A NULL-ra alkalmazott **delete** operátornak nincs semmi hatása. Fontos tudni, hogy a **delete** csak **new**-val létrehozott dinamikus objektumok megszüntetésére használható.

A dinamikus objektum megszüntetése a

```
delete pz;
```

utasítással történik.

A feladat teljes megoldását a MUV_CPP4.CPP tartalmazza:

```
#include <iostream.h>
class Muvelet
{
  private
        float x,y;
        float eredmeny;
        char  muv;
  public:
        Muvelet(char muv1, float x1, float y1);
        ~Muvelet();
        void Szamol();
        void Kiir();
 };

Muvelet::Muvelet(char muv1, float x1, float y1)
{
  muv = muv1;
  x = x1;
  y = y1;
  eredmeny = 0;
}
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
```

```
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
  << eredmeny << endl;
}
void main()
{
 Muvelet *pz;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new Muvelet(muv1,x1,y1);
  pz->Szamol();
  pz->Kiir();
  delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 3
2. adat: 3
3 * 3 = 9
```

3.2.2.12. Dinamikus helyfoglalású objektumpéldányok

Használjunk két dinamikus helyfoglalású objektumpéldányt a műveletek elvégzésére!
A feladat megoldását a MUV_CPP5.CPP állomány tartalmazza.

A két dinamikus objektumpéldány mutatójának definiálása:

```
Muvelet *pz1, *pz2;
```

Az objektumpéldányok létrehozása:

```
pz1 = new Muvelet(muv1,x1,y1);
pz2 = new Muvelet(muv1,x1,y1);
```

illetve megszüntetése:

```
delete pz1;
delete pz2;
```

A feladat teljes megoldása:

```
#include <iostream.h>

class Muvelet
{
  private:
        float x,y;
        float eredmeny;
        char  muv;
  public:
        Muvelet(char muv1, float x1, float y1);
        ~Muvelet();
        void Szamol();
        void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1)
{
  muv = muv1;
  x = x1;
  y = y1;
  eredmeny = 0;
}
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
       << eredmeny << endl;
}

void main()
{
 Muvelet *pz1, *pz2;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz1 = new Muvelet(muv1,x1,y1);
  pz1->Szamol();
  pz1->Kiir();
```

```
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz2 = new Muvelet(muv1,x1,y1);
  pz2->Szamol();
  pz2->Kiir();
  delete pz1; delete pz2;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): -
1. adat: 8.2
2. adat: 2.4
6.2 - 2.4 = 3.8
művelet (+,-,*,/): +
1. adat: 5.2
2. adat: 3.4
5.2 + 3.4 = 8.6
```

3.2.2.13. Objektum adattagjainak dinamikus létrehozása new eljárással

Oldjuk meg a feladatot oly módon, hogy a *Muvelet* osztály dinamikus adattagokat használjon!
A feladat megoldását a MUV_DIN1.CPP állomány tartalmazza.

A *Muvelet* osztály *x,y,eredmeny* adattagjai **float*** mutatók. Az osztály inicializálása során helyet kell foglalni a **float** típusú mutatók számára a memóriában. Ez a feladat a konstruktorra hárul.
A *Muvelet* osztály módosítása:

```
class Muvelet
{
  private:
       float *x,*y;
       float *eredmeny;
       char  *muv;
  public:
       Muvelet(char muv1, float x1, float y1);
       ~Muvelet();
       void Szamol();
       void Kiir();
 };
```

A konstruktor

```
Muvelet::Muvelet(char muv1, float x1, float y1)
{
  x = new float;
  y = new float;
  eredmeny = new float;
  muv = new char;
  *muv = muv1;
  *x = x1;
  *y = y1;
  *eredmeny = 0;
}
```

A mutatóknak a **new** operátor hívásával foglalunk helyet a konstruktorban.

A destruktor feladata a négy mutató által megjelölt területének felszabadítása a **delete** operátorral:

```
Muvelet::~Muvelet()
{
  delete x; delete y;
  delete eredmeny; delete muv;
}
```

A feladat teljes megoldása:

```
#include <iostream.h>
class Muvelet
{
      public:
             float *x,*y;
             float *eredmeny;
             char  *muv;
      public:
             Muvelet(char muv1, float x1, float y1);
             ~Muvelet();
             void Szamol();
             void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1)
{
  x = new float;
  y = new float;
  eredmeny = new float;
  muv = new char;
  *muv = muv1;
  *x = x1; *y = y1;
  *eredmeny = 0;
}
```

```
Muvelet::~Muvelet()
{
  delete x;
  delete y;
  delete eredmeny;
  delete muv;
}
void Muvelet::Szamol()
{
 switch (*muv)
 {
       case '+' : *eredmeny = *x+*y;  break;
       case '-' : *eredmeny = *x-*y;  break;
       case '*' : *eredmeny = *x* *y; break;
       case '/' : *eredmeny = *x/ *y; break;
 }
}
void Muvelet::Kiir()
{
  cout << *x << " " << *muv << " " << *y << " = "
       << *eredmeny << endl;
}

void main()
{
 Muvelet *pz;
 char muv1;
 float x1,y1;
   cout << "művelet (+,-,*,/): "; cin >> muv1;
   cout << "1. adat: "; cin >> x1;
   cout << "2. adat: "; cin >> y1;
   pz = new Muvelet(muv1,x1,y1);
   pz->Szamol();
   pz->Kiir();
   delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 2.5
2. adat: 1.2
2.5 * 1.2 = 3
```

3.3. Öröklés

Mi jut eszünkbe az öröklés szóról? Talán az, hogy a gyermek milyen tulajdonságokat örökölt a szüleitől, vagy gondolhatunk egy családfára is. Hasonló családfa építhető fel osztályokkal is.

Az öröklés (*inheritance*) a C++ legfőbb sajátossága. Ez a mechanizmus teszi lehetővé, hogy bizonyos osztályokból más osztályokat származtassunk, amely során adattagokat és tagfüggvényeket örökíthetünk. Az örökölt tulajdonságok kiterjeszthetők és megváltoztathatók.

A C++ támogatja a többszörös öröklődést (*multiple inheritance*), melynek folyamán egy új osztályt több alaposztályból származtatunk.

A származtatott osztály (*derived class*) olyan osztály, mely az adattagjait és a tagfüggvényeit egy vagy több előzőleg definiált osztálytól örökli. Azt az osztályt, amelytől a származtatott osztály örököl, alaposztálynak (*base class*) nevezzük. A származtatott osztály szintén lehet alaposztálya további osztályoknak, lehetővé téve ezzel egy osztály-hierarchia kialakítását.

Egy származtatott osztály alaposztálya minden tagját örökli, de az alaposztályból csak a **public** és **protected** tagokat éri el. Egy származtatott osztály az öröklött tagokat saját adattagokkal és tagfüggvényekkel egészítheti ki.

A származtatási listában megadott **public** és **private** kulcsszavak – melyeket nem szabad összetéveszteni a tagokhoz való hozzáférés módosítóival (**public**, **private**, **protected**) – az öröklött tagok elérhetőségét szabályozzák. A **public** származtatás során az öröklött tagok megtartják az alaposztálybeli állapotukat, míg **private** származtatás során az öröklött tagok **private** tagokká válnak. (Az alapértelmezés szerinti származtatási mód **class** típusú alaposztály esetén **private**, míg a **struct** típust használva **public**.)

Egy alaposztály "*barátja*" (**friend**) a származtatott osztályban csak az alaposztálytól öröklött tagokat érheti el. Egy származtatott osztály "*barátja*" (**friend**) az alaposztályból csak a **public** és a **protected** tagokat érheti el.

3.3.1. Öröklés az objektum-hierarchiában

Oldjuk meg a feladatot oly módon, hogy az operandusokat az *Adatok* osztály, a műveleti jelet a *MuveletiJel* osztály, az eredményt pedig az *Eredmeny* osztály kezeli és a *Muvelet* osztály örököl az említett osztályoktól és tartalmazza a *Szamol* tagfüggvényt.

A feladat megoldását a MUV_O1.CPP állomány tartalmazza.

Példaként nézzük meg az *Adatok* osztályt, amely az *x* és *y* adattagokat tartalmazza, valamint az osztály konstruktorát!

```
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1, float y1);
    ~Adatok();
    float GetAdatX();
    float GetAdatY();
};
Adatok::Adatok(float x1,float y1)
{
  x = x1;
  y = y1;
}
```

A *MuvetetiJel* osztály a műveleti jelet tárolja és a konstruktora gondoskodik a *muv* adattag kezdőértékéről:

```
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
```

A *Muvelet* osztályt **public** eléréssel származtatjuk a *MuveletiJel*, az *Adatok* és az *Eredmeny* osztályoktól. A származtatás szintaxisa a származtatandó osztály neve után kettőspont (:) majd aszármaztatási mód az osztály nevével.

Többszörös öröklés esetén vesszővel elválasztva újabb osztályokat csatlakoztathatunk a származtatáshoz.

```
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
    public:
           Muvelet(char muv1, float x1, float y1);
           ~Muvelet();
           void Szamol();
           void Kiir();
 };

Muvelet::Muvelet(char muv1, float x1, float y1):
        MuveletiJel(muv1),
        Adatok(x1,y1),
        Eredmeny()
{
}
```

A konstruktor paraméterlistáján szerepelnie kell az összes kezdőértékadásához szükséges paramétereknek, mert ezek egy részét tovább kell adni az ős osztályoknak (MUV_O1.CPP):

```
#include <iostream.h>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1, float y1);
    ~Adatok();
    float GetAdatX();
    float GetAdatY();
};
Adatok::Adatok(float x1,float y1)
{
  x = x1;
  y = y1;
}

Adatok::~Adatok(){ }
float Adatok::GetAdatY()
{
  return y;
}
```

```
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
class Eredmeny
{
  protected:
     float eredmeny;
  public:
     Eredmeny();
};
Eredmeny::Eredmeny()
{
  eredmeny = 0;
}
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
     public:
            Muvelet(char muv1, float x1, float y1);
            ~Muvelet();
            void Szamol();
            void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1):
          MuveletiJel(muv1),
          Adatok(x1,y1),
          Eredmeny()
{
}
Muvelet::~Muvelet()
{
      // üres a származtatott osztály konstruktorának a törzse
}
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
```

```
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
       << eredmeny << endl;
}
void main()
{
 Muvelet *pz;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new Muvelet(muv1,x1,y1);
  pz->Szamol();
  pz->Kiir();
  delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): +
1. adat: 3.2
2. adat: 4.5
3.2 + 4.5 = 7.7
```

3.3.2. Objektum öröklése

Definiáljuk egy *SajaMuvelet* osztályt, amely a *Muvelet* osztályból származik!
A feladat megoldását a MUV_OWN.CPP állomány tartalmazza.

A *SajatMuvelet* osztály mindent örököl a *Muvelet* osztálytól, így a *Szamol* tagfüggvénnyel is számoltathat.

A *SajatMuvelet* osztály:

```
class SajatMuvelet:public Muvelet
{
  public:
    SajatMuvelet(char,float,float);
    ~SajatMuvelet();
};
SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
              Muvelet(muv1,x1,y1)
{
}
SajatMuvelet::~SajatMuvelet()
{
}
```

A feladat megoldását a MUV_OWN.CPP állomány tartalmazza:

```
#include <iostream.h>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1,float y1);
};
Adatok::Adatok(float x1, float y1)
{
  x = x1;  y = y1;
}
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
class Eredmeny
{
  protected:
     float eredmeny;
  public:
     Eredmeny();
};
Eredmeny::Eredmeny()
{
  eredmeny = 0;
}
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
  public:
       Muvelet(char muv1, float x1, float y1);
       ~Muvelet();
       void Szamol();
       void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1):
         MuveletiJel(muv1),
         Adatok(x1,y1),
         Eredmeny()
{ }
```

```
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
      case '+' : eredmeny = x+y; break;
      case '-' : eredmeny = x-y; break;
      case '*' : eredmeny = x*y; break;
      case '/' : eredmeny = x/y; break;
 }
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
     << eredmeny << endl;
}
class SajatMuvelet:public Muvelet
{
  public:
    SajatMuvelet(char,float,float);
    ~SajatMuvelet();
};
SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
              Muvelet(muv1,x1,y1)
{ }
SajatMuvelet::~SajatMuvelet()
{ }
void main()
{
 SajatMuvelet *pz;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new SajatMuvelet(muv1,x1,y1);
  pz->Szamol();
  pz->Kiir();
  delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 4
2. adat: 5
4 * 5 = 20
```

3.3.3. Objektum öröklése a *Szamol* tagfüggvény újradefiniálásával

A *SajatMuvelet* osztálynak definiáljunk *Szamol* tagfüggvényt, amely a négy alapműveleten kívül a pozitív alapú hatványozást is elvégzi.

A feladat megoldását a MUV_OWN2.CPP állomány tartalmazza.

A *SajatMuvelet* osztály *Szamol* tagfüggvényében a négy alapművelet a hatványozással bővült.

```
void SajatMuvelet::Szamol()
{
 switch (muv)
 {
  case '+' : eredmeny = x+y; break;
  case '-' : eredmeny = x-y; break;
  case '*' : eredmeny = x*y; break;
  case '/' : eredmeny = x/y; break;
  case '^' : if( x > 0) eredmeny = pow(x,y);
             else eredmeny = 0; break;
 }
}
```

Definiáljuk az alábbi objektumpéldányt:

```
SajatMuvelet  *pz;
```

Hozzuk létre és inicializáljuk:

```
pz = new SajatMuvelet(muv1,x1,y1);
```

Ha meghívjuk a *Szamol* tagfüggényt, akkor a sajátja aktivizálódik:

```
cout << x << " " << m << " " << y << " = " << psz->Szamol();
```

A feladat megoldását a MUV_OWN2.CPP tartalmazza:

```
#include <iostream.h>
#include <math.h>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1,float y1);
};
Adatok::Adatok(float x1, float y1)
{
  x = x1;
  y = y1;
}
```

```
class MuveletiJel
{
  protected:
   char muv;
  public:
   MuveletiJel(char muv1);
};

MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
class Eredmeny
{
  protected:
     float eredmeny;
  public:
     Eredmeny();
};
Eredmeny::Eredmeny()
{
  eredmeny = 0;
}
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
  public:
      Muvelet(char muv1, float x1, float y1);
      ~Muvelet();
      void Szamol();
      void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1):
        MuveletiJel(muv1),
        Adatok(x1,y1),
        Eredmeny()
{ }
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
 }
}
```

```
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
       << eredmeny << endl;
}
class SajatMuvelet:public Muvelet
{
  public:
    SajatMuvelet(char,float,float);
    ~SajatMuvelet();
    void Szamol();
};
SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
              Muvelet(muv1,x1,y1)
{ }
SajatMuvelet::~SajatMuvelet()
{ }
void SajatMuvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; break;
   case '-' : eredmeny = x-y; break;
   case '*' : eredmeny = x*y; break;
   case '/' : eredmeny = x/y; break;
   case '^' : if( x > 0) eredmeny = pow(x,y);
              else eredmeny = 0; break;
 }
}
void main()
{
 SajatMuvelet *pz;
 char muv1;
 float x1,y1;
   cout << "művelet (+,-,*,/,^): "; cin >> muv1;
   cout << "1. adat: "; cin >> x1;
   cout << "2. adat: "; cin >> y1;
   pz = new SajatMuvelet(muv1,x1,y1);
   pz->Szamol();
   pz->Kiir();
   delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/,^): ^
1. adat: 2
2. adat: 3
2 ^ 3 = 8
```

3.3.4. Az objektumok zártsága

Bővítsük a *Muvelet* osztályt a *VegreHajt* tagfüggvénnyel, amelyben a *Szamol* tagfüggvényt hívjuk!
A feladat megoldását a MUV_OWN3.CPP állomány tartalmazza.

A *VegreHajt* tagfüggvény aktiválja a *Szamol* tagfüggvényt:

```
void Muvelet::VegreHajt()
{
  Szamol();
}
```

A *pz* mutatóval kijelölt dinamikus objektumpéldány aktiválja az öröklött *Vegrehajt* tagfüggvényt

```
        pz->VegreHajt();
```

Nem a várt eredményt kaptuk!

```
#include <iostream.h>
#include <math.h>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1,float y1);
};
Adatok::Adatok(float x1, float y1)
{
  x = x1;
  y = y1;
}
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
```

```
class Eredmeny
{
  protected:
     float eredmeny;
  public:
     Eredmeny();
};
Eredmeny::Eredmeny()
{
  eredmeny = 0;
}
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
  public:
       Muvelet(char muv1, float x1, float y1);
       ~Muvelet();
       void Szamol();
       void Vegrehajt();
       void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1):
         MuveletiJel(muv1),
         Adatok(x1,y1),
         Eredmeny()
{ }
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; Kiir(); break;
   case '-' : eredmeny = x-y; Kiir(); break;
   case '*' : eredmeny = x*y; Kiir(); break;
   case '/' : eredmeny = x/y; Kiir(); break;
   default: cout << "Hibás műveleti jel! \n";
 }
}
void Muvelet::Vegrehajt()
{
  Szamol();
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = "
       << eredmeny << endl;
}
```

```
class SajatMuvelet:public Muvelet
{
 public:
   SajatMuvelet(char,float,float);
   ~SajatMuvelet();
   void Szamol();
};
SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
                Muvelet(muv1,x1,y1)
{ }
SajatMuvelet::~SajatMuvelet()
{}
void SajatMuvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; Kiir(); break;
   case '-' : eredmeny = x-y; Kiir(); break;
   case '*' : eredmeny = x*y; Kiir(); break;
   case '/' : eredmeny = x/y; Kiir(); break;
   case '^' : if( x > 0) { eredmeny = pow(x,y); Kiir(); }
              else { eredmeny = 0; Kiir(); } break;
   default: cout << "Hibás műveleti jel! \n";
 }
}
void main()
{
 SajatMuvelet *pz;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/,^): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new SajatMuvelet(muv1,x1,y1);
  pz->Vegrehajt();
  delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/,^): ^
1. adat: 2
2. adat: 3
Hibás műveleti jel!
```

Nem a várt eredményt kapjuk, hiszen nem "hívódott" meg a *SajatMuvelet* osztály *Szamol* tagfüggvénye, ezért hatványozási műveleti jel olvasásakor "Hibás műveleti jel" hibajelzés jelenik meg. A jelenség oka az objektumok zártsága. A zártság elvének érvényesüléséről a fordítóprogram azzal gondoskodik, hogy a tagfüggvényekből elvégzett tagfüggvényhívásokat a hagyományos függvényhívásokkal azonos módon (rögzített címmel) fordítja.

Az objektum-orientált nyelvekben az ilyen tagfüggvényeket statikus (rögzített hivatkozású) tagfüggvényeknek nevezzük. E probléma feloldására a következő részben látunk példát.

3.4. Sokalakúság (polimorfizmus)

Az objektum-orientált programozás további fontos jellemzője a sokalakúság (*polymorphism*), amely lehetővé teszi, hogy egy adott őstípusból származtatott további típusok természetesen öröklik az őstípus minden mezőjét, így a tagfüggvényeket is. Az öröklés során a tulajdonságok egyre módosulhatnak, azaz például egy öröklött tagfüggvény nevében ugyan nem változik egy leszármazottban, de esetleg már egy kicsit (vagy éppen nagyon) másképp viselkedik. Ezt a C++-ban a legflexibilisebb módon az ún. virtuális függvények (*virtual functions*) teszik lehetővé.

A sokalakúság akkor érvényesül, ha virtuális tagfüggvényeket használunk. A virtuális tagfüggvények biztosítják, hogy egy adott osztály-hierarchiában (származási fán) egy adott függvény különböző verziói létezzenek úgy, hogy csak a kész program futása során derül ki, hogy ezek közül éppen melyiket kell végrehajtásra meghívni. Ezt a mechanizmust, azaz a hívó és a hívott függvény futási idő alatt történő összerendelését késői kötésnek (*late binding*) nevezzük. A fordítás során megvalósított összerendelést korai kötésnek (*early binding*) hívjuk.

A konstruktor kivételével minden tagfüggvény (beleértve a destruktorokat is) virtuálissá tehető a **virtual** direktíva megadásával.

3.4.1. Virtuális tagfüggvények

A virtuális függvények deklarálásának szintaktikája nagyon egyszerű: a tagfüggvény első deklarálásakor elhelyezzük a **virtual** típusmódosító szót:

Figyelem! Csak tagfüggvények deklarálhatók virtuális függvényként. Ha egy függvényt egyszer már virtuálisnak deklaráltunk, akkor egyetlen származtatott osztályban sem deklarálhatjuk újra az adott függvényt ugyanilyen paraméter-szignatúrával, de más visszatérési típussal. Ha egy virtuális függvényt valamely származtatott típusban újradeklarálunk és az újbóli deklaráció alkalmával ugyanolyan visszatérési típust és paraméter-szignatúrát alkalmazunk, mint a

korábbi deklarációban, akkor az újonnan deklarált függvény automatikusan virtuális lesz.

Tehát, ha egy függvényt az alaposztályban virtuálisként deklarálunk, akkor ezt a tulajdonságát az öröklődés során is megőrzi. A származtatott osztályban a virtuális függvényt saját változattal újradefiniálhatjuk, de az öröklött verziót is használhatjuk. Saját verzió definiálásakor nem szükséges a **virtual** szót megadnunk.

A *Muvelet* osztály *Szamol* tagfüggvényét tegyük virtuálissá!
A feladat megoldását a MUV_OWN4.CPP állomány tartalmazza.

```
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
  public:
      Muvelet(char muv1, float x1, float y1);
      ~Muvelet();
      virtual void Szamol();
      void Vegrehajt();
      void Kiir();
 };
```

Elegendő csak a *Szamol* függvényt virtuálissá tenni.

```
#include <iostream.h>
#include <math.h>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(float x1,float y1);
};

Adatok::Adatok(float x1, float y1)
{
  x = x1;
  y = y1;
}
```

```
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};

MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
class Eredmeny
{
  protected:
     float eredmeny;
  public:
     Eredmeny();
};
Eredmeny::Eredmeny()
{
  eredmeny = 0;
}
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{
  public:
       Muvelet(char muv1, float x1, float y1);
       ~Muvelet();
       virtual void Szamol();
       void Vegrehajt();
       void Kiir();
 };
Muvelet::Muvelet(char muv1, float x1, float y1):
         MuveletiJel(muv1),
         Adatok(x1,y1),
         Eredmeny()
{ }
Muvelet::~Muvelet()
{ }
void Muvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; Kiir(); break;
   case '-' : eredmeny = x-y; Kiir(); break;
   case '*' : eredmeny = x*y; Kiir(); break;
   case '/' : eredmeny = x/y; Kiir(); break;
   default: cout << "Hibás műveleti jel! \n";
 }
}
```

```
void Muvelet::Vegrehajt()
{
  Szamol();
}
void Muvelet::Kiir()
{
  cout << x << " " << muv << " " << y << " = " << eredmeny << endl;
}
class SajatMuvelet:public Muvelet
{
  public:
    SajatMuvelet(char,float,float);
    ~SajatMuvelet();
    void Szamol();
};
SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
              Muvelet(muv1,x1,y1)
{ }
SajatMuvelet::~SajatMuvelet()
{ }
void SajatMuvelet::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; Kiir(); break;
   case '-' : eredmeny = x-y; Kiir(); break;
   case '*' : eredmeny = x*y; Kiir(); break;
   case '/' : eredmeny = x/y; Kiir(); break;
   case '^' :
             if( x > 0) { eredmeny = pow(x,y); Kiir(); }
             else
             if ( x == 0) { eredmeny = 0; Kiir(); }
             else {  cout << "negativ alap\n";  break; }
 }
}
void main()
{
 SajatMuvelet *pz;
 char muv1;
 float x1,y1;
   cout << "művelet (+,-,*,/,^): "; cin >> muv1;
   cout << "1. adat: "; cin >> x1;
   cout << "2. adat: "; cin >> y1;
   pz = new SajatMuvelet(muv1,x1,y1);
   pz->Vegrehajt();
   delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/,^): ^
1. adat: 2
2. adat: 3
2 ^ 3 = 8
```

A *SajatMuvelet::VegreHajt* tagfüggvényben a virtuális *Szamol* tagfüggvényre való hivatkozás a *SajatMuvelet::Szamol* tagfüggvény végrehajtását eredményezi.

3.4.2. Osztály típusú adattagra mutató pointer

Oldjuk meg a feladatot oly módon, hogy az operandust az *Adat* osztály, a műveleti jelet a *MuveletiJel* osztály, az eredményt pedig az *Eredmeny* osztály kezeli és a *Muvelet* osztály adattagjai az *Adat* osztályra, a *MuveletiJel* osztályra és az *Eredmeny* osztályra mutatnak.
A feladat megoldását a MUV_O3.CPP állomány tartalmazza.

Az *Adat* osztály deklarációja:

```
class Adat
{
  protected:
    float w;
  public:
    Adat(float w1);
    ~Adat();
    float GetAdat();
};
Adat::Adat(float w1)
{
  w = w1;
}
Adat::~Adat(){ }
```

A *GetAdat* tagfüggvény a *w* tadattagot adja vissza:

```
float Adat::GetAdat()
{
  return w;
}
```

```
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
    ~MuveletiJel();
    char GetMuveletiJel();
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
MuveletiJel::~MuveletiJel()
{ }
```

A *GetMuv* tagfüggvény a műveleti jelet adja vissza:

```
char MuveletiJel::GetMuveletiJel()
{
  return muv;
}
```

A *Muvelet* osztály az *Eredmeny* osztályból származtatva:

```
class Muvelet: public Eredmeny
{
  protected:
     Adat *px, *py;
     MuveletiJel *pmuv;
  public:
      Muvelet(char muv1, float x1, float y1);
      ~Muvelet();
      void Szamol();
      void Kiir();
 };
```

A konstruktorban, az adattagokban létrejönnek az osztályok:

```
Muvelet::Muvelet(char muv1, float x1, float y1)
{
  px = new Adat(x1);
  py = new Adat(y1);
  pmuv = new MuveletiJel(muv1);
  eredmeny = 0;
}
```

A destruktorban megszünnek a dinamikusan létrehozott objektumok:

```
Muvelet::~Muvelet()
{
  delete px; delete py; delete pmuv;
}
```

A *Szamol* tagfüggvényben a műveleti jelet a *GetMuv*, az operandusokat a *GetAdat* tagfüggvényekkel kérdezzük le:

```
void Muvelet::Szamol()
{
 switch (pmuv->GetMuveletiJel())
 {
   case '+' : eredmeny = px->GetAdat() + py->GetAdat(); break;
   case '-' : eredmeny = px->GetAdat() - py->GetAdat(); break;
   case '*' : eredmeny = px->GetAdat() * py->GetAdat(); break;
   case '/' : eredmeny = px->GetAdat() / py->GetAdat(); break;
 }
}
```

A *Muvelet* osztály *Kiir* tagfüggvénye:

```
void Muvelet::Kiir()
{
  cout << px->GetAdat() << " " << pmuv->GetMuveletiJel()
       << " " << py->GetAdat() << " = " << eredmeny << endl;
}
```

A főprogram:

```
void main()
{
 Muvelet *pz;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new Muvelet(muv1,x1,y1);
  pz->Szamol();
  pz->Kiir();
  delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/): *
1. adat: 2.5
2. adat: 3.2
2.5 * 3.2 = 8
```

3.4.3. A sablonok alkalmazása öröklött objektumok esetén

A MUV_TMP1.CPP program bemutatja a sablonok (*template*-k) alkalmazását öröklött objektumok esetén.

```
/* muv_tmp1.cpp */
#include <iostream.h>
// template: Adatok

template <class R>
class Adatok
{
  protected:
    float x,y;
  public:
    Adatok(R x1,R y1);
};


template <class R>
Adatok<R>::Adatok(R x1, R y1)
{
  x = x1;
  y = y1;
}
class MuveletiJel
{
  protected:
    char muv;
  public:
    MuveletiJel(char muv1);
};
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}
template <class R>
class Eredmeny
{
  pu:
     float eredmeny;
  public:
     Eredmeny();
};
template <class R>
Eredmeny<R>::Eredmeny()
{
  eredmeny = 0;
}
```

```
template <class R>
class Muvelet:public MuveletiJel,
              public Adatok<R>,
              public Eredmeny<R>
{
  public:
       Muvelet(char muv1, R x1, R y1);
       Muvelet();
       ~Muvelet();
       virtual void Szamol();
       void Vegrehajt();
       void Kiir();
 };
template <class R>
Muvelet<R>::Muvelet(char muv1, R x1, R y1):
         MuveletiJel(muv1),
         Adatok<R>(x1,y1),
         Eredmeny<R>()
{ }
template <class R>
Muvelet<R>::~Muvelet()
{ }
template <class R>
void Muvelet<R>::Szamol()
{
 switch (muv)
 {
   case '+' : eredmeny = x+y; Kiir(); break;
   case '-' : eredmeny = x-y; Kiir(); break;
   case '*' : eredmeny = x*y; Kiir(); break;
   case '/' : eredmeny = x/y; Kiir(); break;
   default: cout << "Hibás műveleti jel! \n";
 }
}
template <class R>
void Muvelet<R>::Vegrehajt()
{
  Szamol();
}
template <class R>
void Muvelet<R>::Kiir()
{
  cout << x << " " << muv << " " << y << " = " << eredmeny << endl;
}
template <class R>
class SajatMuvelet:public Muvelet<R>
{
  public:
    SajatMuvelet(char,R,R);
    ~SajatMuvelet();
    void Szamol();
};
```

```
template <class R>
SajatMuvelet<R>::SajatMuvelet(char muv1, R x1, R y1):
              Muvelet<R>(muv1,x1,y1)
{ }
template <class R>
SajatMuvelet<R>::~SajatMuvelet()
{ }
template <class R>
void SajatMuvelet<R>::Szamol()
{
 switch (muv)
 {
  case '+' : eredmeny = x+y; Kiir(); break;
  case '-' : eredmeny = x-y; Kiir(); break;
  case '*' : eredmeny = x*y; Kiir(); break;
  case '/' : eredmeny = x/y; Kiir(); break;
  case '^' :if( x > 0) { eredmeny = exp(y*log(x)); Kiir(); }
            else if ( x == 0) { eredmeny = 0; Kiir(); }
            else {  cout << "negativ alap\n";  break; }
 }
}
void main()
{
 SajatMuvelet <float>*pz;
 SajatMuvelet <int>*pzi;
 char muv1;
 float x1,y1;
  cout << "művelet (+,-,*,/,^): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pz = new SajatMuvelet<float>(muv1,x1,y1);
  cout.setf(iostream::fixed, iostream::floatfield);
  pz->Vegrehajt();
  delete pz;

  cout << "művelet (+,-,*,/,^): "; cin >> muv1;
  cout << "1. adat: "; cin >> x1;
  cout << "2. adat: "; cin >> y1;
  pzi = new SajatMuvelet<int>(muv1,x1,y1);
  pzi->Vegrehajt();
  delete pzi;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/,^): +
1. adat: 2.1
2. adat: 4.3
2.1 + 4.3 = 6.4
művelet (+,-,*,/,^): +
1. adat: 2
2. adat: 3
2 + 3 = 5
```

3. 5. További lehetőségek a C++-ban

Az eddigiekben a C++ leglényegesebb vonásait mutattuk be. Ebben az alfejezetben további, a nyelvet igen rugalmassá tevő tulajdonságokat tekintjük át.

3.5.1. Rokonok és barátok

Az öröklés következtében különböző őstípusokból kiindulva teljes származási fákat, másképp kifejezve osztály-hierarchiákat alakíthatunk ki. A többszörös öröklés azt is lehetővé teszi, hogy egy újabb típust több őstípus leszármazottjaként hozzunk létre. Ily módon egymással rokonságban álló típusok definiálhatók. Elképzelhető azonban, hogy egy programon belül egymástól teljesen független családfákat hozunk létre. Ugyanazon típuscsaládba (származási fába, családfába, osztályhierarchiába) tartozó típusok egymás rokonai, míg a különböző családba tartozók egymás számára teljesen idegenek.

Az életben az ember természetesen nemcsak a rokonaival óhajtja tartani a kapcsolatot, hanem szüksége van barátokra is, azaz olyan egyedekre, akikkel nincs semmiféle származási kapcsolatban, de mégis fontosak egymás számára. Nos, egy C++ programban deklarált típusok esetében is hasonló a helyzet. Ha komolyan vettük az információrejtést, szigorúan kézben tartottuk a taghozzáférést az egyes típuscsaládok definíciójakor, akkor egy adott típus egy tagfüggvényei nem férhetnek hozzá másik – nem rokon – osztály adattagjaihoz. Márpedig nem zárható ki az az eset, amikor az ilyen hozzáférés haszonnal járhat. Hogy egy C++ programban ezt a hasznot élvezhessük, a **friend** kulcsszóval megadhatjuk, hogy a programok nem rokon tárolási egységei közül melyek között van baráti viszony.

Kicsit egzaktabban: szükségünk lehet arra, hogy egy adott osztály privát adattagjait egy olyan függvénnyel is kezelhessük, amely nem eleme az adott osztálynak. Ezt úgy érhetjük el, hogy az osztály deklarációjában a **friend** kulcsszóval szerepeltetjük azon külső függvénye prototípusát, amelyek számára jogot szeretnénk biztosítani az adott osztály adatmezőinek elérésére. A **friend** deklaráció az osztály-deklarációnak bárhol elhelyezhető. Lássunk néhány példát a "függvény-barátságra".

```
class xx
{
   protected:
        int data_1;
   private:
        int data_2;
        friend int ext_func(void);
   public:
        xx(int d1, int d2);
        int get_data_1(void);
};
int ext_func(void) { ... }
...
```

Az előbbi példában *ext_func()* egy normál C++ függvény. Az *xx* osztály deklarációjakor jogot biztosíthattunk számára az adattagok (*data_1*, *data_2*) elérésére. Természetesen egy másik osztály tagfüggvényei is lehetnek barátok:

```
class yy
{
     . . .
     void yy_func1(void);
     . . .
};
class zz
{
     . . .
     void zz_func(void);
     friend void yy::yy_func1(void)1
     . . .
};
```

Itt tehát az *yy* osztály *yy_func1* nevű tagfüggvényét deklaráltuk a *zz* osztály barátjának. Arra is van lehetőségünk, hogy az *yy* osztályból ne csak egy, hanem minden tagfüggvényt a *zz* osztály barátjává tegyünk:

```
class zz
{
     . . .
     void zz_func(void);
     friend class yy;
     . . .
};
```

Ez a barátság egyoldalú, hiszen az *yy* osztály definíciós blokkjában a *zz* osztályt nem deklaráltuk barátként. A "függvény barátságra" vonatkozó részletes mintapéldákat az ún. operátor-függvények kapcsán mutatunk be a következő alfejezetekben.

3.5.2. Függvények értelmezésének kiterjesztése (*function overloading*)

A C++-nak van egy speciális tulajdonsága, amellyel kevés más nyelv rendelkezik. Nevezetesen, a nyelv által definiált, létező operátorokhoz újabb jelentést rendelhetünk. Ez hasznos lehet abból a célból, hogy valamilyen osztályként definiált új adatstruktúrán az elemi adattípusok esetében megszokott műveletekhez hasonló műveleteket ugyanahhoz a műveleti jelhez rendelhessünk, amit az elemi típusoknál megszoktunk. Ha például definiálunk egy halmazok reprezentációjára szolgáló típust, akkor nagyon kellemes, ha a + operátorral halmazok unióját, a * operátorral pedig halmazok metszetét képezhetjük. Vagy ha *a* és *b* egy-egy mátrix típusú objektum, akkor jó lenne, ha az *a*b* kifejezés szintén mátrix típusú eredményt, nevezetesen a két mátrix szorzatát szolgáltatná.

A polimorfizmus tárgyalásakor már utaltunk arra, hogy egy adott művelet végrehajtását jelentő szimbólumhoz többféle jelentést is rendelhetünk. Ez akkor hasznos, ha ugyanazt a műveletet különböző típusú adatokon akarjuk végrehajtani. Ennek a legegyszerűbb esete az ún. függvény *overloading*. Tekintsük a következő példát:

```
void muvelet (int &a, int &b, int &c, char jel)
{
   cout << "int " << jel <<  " int verzió";
   switch (jel)
   {
      case '+': c = a+b; break;
      case '-': c = a-b; break;
      case '*': c = a*b; break;
      case '/': c = a/b; break;

   }
}

void muvelet (double &a, double &b, double &c, char jel)
{
   cout << "double " << jel << " double verzió";
   switch (jel)
   {
      case '+': c = a+b; break;
      case '-': c = a-b; break;
      case '*': c = a*b; break;
      case '/': c = a/b; break;

   }
}
```

Egy hagyományos C fordító a fenti két függvénydefiníciók esetében hibajelzést adna, mondván, hogy a *muvelet()* függvényt kétszer definiáltuk. Azonban egy C++ fordító a fenti függvénydefiníciókat két különböző függvénynek tekinti, mert a formális paraméterlistájuk, az ún. *paraméter-szignatúrájuk* különböző. Azaz úgy tekinti, hogy van egy *muvelet(**int** &, **int** &, **int** &, **char**)* függvényünk és egy másik, *muvelet(**double** &, **double** &, **double** &, **char**)* függvényünk. Ekkor a

```
/* muv.cpp */
void main()
{
  double x = 1.5, y = 2.3, z;
  int    i = 30,  j = 40, k;

  cout << endl;
  cout << i << " + " << j << " művelet, ";
  muvelet(i,j,k,'+');
  cout << ", eredménye: " << k << endl;
  cout << x << " + " << y << " művelet, ";
  muvelet(x,y,z,'+');
  cout << ", eredménye: " << z << endl;
}
```

függvényhívások esetében a fordító az *aktuális paraméterlista* alapján tudja eldönteni, hogy a *muvelet()* függvény melyik változatát kell aktivizálni. Első esetben, amikor csupa egész típusú adatot adunk meg, a paraméter-szignatúra megfeleltetés alapján tehát a függvény *muvelet(**int** &, **int** &, **int** &, **char**)* változata, második esetben, amikor csupa **double** típusú változóval aktivizáljuk a függvényt, a *muvelet(**double** &, **double** &, **double** &, **char**)* paraméter-szignatúrájú változata kerül aktivizálásra.

A fenti program (MUV.CPP) futtatása után a következő eredményt kapjuk:

```
30 + 40 művelet, int + int verzió, eredménye: 70
1.5 + 2.3 művelet, double + double verzió, eredménye: 3.8
```

Egyszerű példánk alapján tehát azt mondhatjuk, hogy az *overloading* fogalma azt takarja, amikor egyazon szimbólumhoz - a *muvelet* függvényazonosítóhoz - többféle értelmezést adunk meg. Azt is mondhatjuk, hogy az első függvénydefiníciónkat - *muvelet(**int** &, **int** &, **int** &, **char**)* - *átdefiniáltuk* a másodikkal. Helyesebb azonban inkább azt mondani, hogy bővítettük a *muvelet* szimbólum jelentését, illetve *kiterjesztettük az értelmezését*. (Az *overloading* szó szerinti fordításban *túlterhelést* jelent, ami egy angol anyanyelvű számára szemléletes:

a függvényazonosítót egy újabb jelentéssel terheltük meg. A lényeget azonban talán jobban kifejezi az *értelmezés kiterjesztése.*)

A *függvények sokalakúságának* leggyakrabban előforduló példája az, amikor egy osztálydefiníció során több, különböző paraméter-szignatúrájú konstruktort adunk meg.

3.5.3. Operátorok értelmezésének kiterjesztése (*operator overloading*)

Az *operator overloading* fogalma szorosan kötődik a függvény *overloading*-hoz. Itt is arról van tehát szó, hogy ugyanazt a műveletet több, különböző típusú adaton is végre szeretnénk hajtani. Valójában a szabványos C++ operátorokhoz előre definiált ún. *operátor-függvények* tartoznak. Amikor tehát egy C++-ban definiált műveleti jel értelmezését ki szeretnénk terjeszteni egy saját adattípusunkra, akkor a kérdéses operátor operátor-függvényére adunk meg egy újabb paraméter-szignatúrájú változatot. Ez azt jelenti, hogy az operátor *overloading* a függvény *overloading* speciális esete. (Gondoljunk bele, hogy az operator *overloading* nem egészen C++ tulajdonság, hiszen a hagyományos C-beli operátorok is képesek különböző adattípusokon is ugyanazt a műveletet végrehajtani. Erre egy igen kézenfekvő példa az értékadás operátor, amelyik mind különböző sorszámozott típusú adatokon, mind pedig lebegőpontos számokon értelmezett művelet, stb.)

Tehát az osztályok tervezésekor felmerülhet annak az igénye, hogy a kérdéses osztály objektumai számára olyan műveleteket definiáljunk, amelyeket a szokásos C++ operátorokhoz rendelünk, azok értelmezésének kiterjesztése által, amely a szükséges operátor-függvény elkészítésével történik. Egy operátor-függvény a kérdéses objektumosztály tagfüggvénye, de legalábbis “barátja” kell legyen, és legalább egy olyan argumentumának kell lenni, amelynek típusa a kérdéses osztály. Ez utóbbi szabály azért fontos, mert ez akadályozza meg azt, hogy egy műveleti jel eredeti, a C++ nyelv előre definiált típusaira vonatkozó értelmezését felülbíráljuk. Egy operátor-függvényt pontosan ugyanúgy kell definiálni, mint egy közönséges függvényt, azzal a különbséggel, hogy a definícióban a szokásos függvényazonosító helyett az **operator** kulcsszó és azt követően egy C++ operátor álljon. A 3.1. táblázat mutatja be az átdefiniálható, kiterjeszthető C++ operátorokat.

3.1. táblázat

+	-	*	/	%	^	&	|
~	!	,	=	<	>	<=	>=
++	--	<<	>>	==	!=	&&	||
+=	-=	/=	%=	^=	&=	|=	<<=
>>=	[ ]	( )	->	->*	**new**	**delete**	

Egy operátor-függvény általános alakja tehát:

```
osztály::osztály& operator jel (paraméterlista) { }
```

ha az *osztály* típus tagfüggvényeként definiáljuk az operátor-függvényt. Itt *jel* a fenti, 3.1. táblázatban felsorolt operátorok egyike. Az ilyen definíció bizonyos operátorok esetében (mint például az értékadás) kötelező. Sokszor azonban az osztály megadásakor **friend**-ként is definiálhatunk operátor-függvényt:

```
friend típus operator jel (paraméterlista) { }.
```

Példaként definiáljunk egy osztályt a halmazalgebrai műveletek megvalósításához. A létrehozandó *halmaz* típus, az egyszerűség kedvéért legyen az egész számok halmaza. Az ehhez szükséges minimális osztálydefiníció a következő:

```
class halmaz
{
    int  *elemek;                          // elemek tömbje
    int  darab;                            //halmaz számossága
public:
     halmaz(int n=0, int *e=NULL);              // konstruktor
     ~halmaz(){if (darab) delete [darab] elemek;} // destruktor
};

halmaz::halmaz(int n, int *e)
{
  int i;

  if (n)
    for (i = 0, darab = n, elemek = new int[n]; i < n; i++)
       elemek[i] = e[i];
  else
  {
    darab = 0;
    elemek = NULL;
  }
}
```

A halmaz aktuális elemeit az *elemek* mutatóval megcímzett dinamikusan létrehozott egész tömbben tároljuk, a halmaz számosságát pedig a *darab* mezőben tartjuk nyilván. Definiáltunk egy konstruktort, amely paraméterként egy egész számot és egy egész tömböt vár, feladata pedig az elemek tárolására szolgáló tömb létrehozása és feltöltése a paraméterként átadott tömb elemeivel. Az elemek számát a konstruktor első paramétere adja meg. Ha a konstruktort paraméter nélkül aktivizáljuk, akkor üres halmaz jön létre, így az `elemek` pointer értéke NULL lesz. A destruktor nagyon egyszerű: a nem üres halmaz elemeit tároló, dinamikusan foglalt memóriaterületet felszabadítja, míg üres halmaz esetén nincs teendője. Figyeljük meg, hogy mivel nagyon egyszerű a destruktor-függvény, az *inline* megvalósítást választottuk. Ugyancsak érdemes megemlíteni, hogy a **delete** operátor összetett alakját alkalmaztuk annak érdekében, hogy az *elemek* mutató által megcímzett memóriaterületet egy *darab* méretű tömbként felszabadítsa. Az osztálydefiníció alapján tehát a következőképpen definiálhatunk halmaz-objektumokat:

```
int szamok[] = {1,2,3,4,5}; // Ilyen elemekkel töltünk fel egy halmazt

halmaz a(sizeof(szamok)/sizeof(int),szamok);  // Elemei: 1,2,3,4,5
halmaz b;                                     // Üres halmaz
```

Az *a* halmaz 5 egész számot fog tartalmazni, melyeket a *szamok* tömbben adunk meg. A halmaz számosságának kiszámítását a C++ fordító végzi el a ***sizeof*** operátort tartalmazó kifejezés kiértékelésével. A *b* halmaz üres halmaz lesz. Eddigi definíciónk alapján fenti halmazaink nem módosíthatók, például a *b* halmaz mindig üres halmaz marad. Nyilvánvaló, hogy az egyik legfontosabb művelet, amelynek értelmezését az imént definiált *halmaz* típusra ki kell terjesztenünk, az értékadás művelete. Az alábbiakban megadunk egy olyan operátor-függvényt, amely az = operátort (az értékadás műveletét) kiterjeszti a *halmaz* típusra is:

```
halmaz& halmaz::operator=(const halmaz &s)
{
  int i;
  darab = s.darab;
  if (!s.darab) { elemek = NULL; }
  else
  {
    elemek = new int[darab];
    for (i = 0; i < darab; i++) elemek[i] = s.elemek[i];
  }
 return *this;
}
```

Így minden olyan esetben, amikor egy halmazt egy másik halmaznak értékül adunk, ez az operátor-függvény kerül meghívásra, tehát a fenti *b* halmaz is kaphat értéket ennek segítségével:

```
...
void main(void)
{
  b = a;  // A halmaz::operator= függvény aktivizálása.
}
```

Egy osztály operátor-függvényei is lehetnek többértelműek, feltéve, hogy paraméter-szignatúrájuk egyedi. Például a *halmaz* típus esetén értelmezhetünk egy másik értékadást is, amellyel egyetlen elemet tartalmazó halmazok hozhatók létre:

```
class halmaz
{
     int  *elemek;                              // elemek tömbje
     int  darab;                                // halmaz számossága
public:
      halmaz(int n=0, int *e=NULL);             // konstruktor
      ~halmaz(){ if (darab) delete elemek; }// destruktor
      halmaz& operator=(halmaz &s);        // halmaz = halmaz értékadás
      halmaz& operator=(int i);            // halmaz = egész értékadás
};
```

3.5.4. Operátor-függvények definiálása

A C++ operátorok közül csak az 3.1. táblázatban található operátorok jelentése terjeszthető ki, alkalmazható rájuk az *operator overloading*. Egy osztály tervezésekor tehát új operátor (például `**` a hatványozásra) nem definiálható. Ugyancsak nem változtatható meg az operátoroknak a C++ előre definiált típusaira vonatkozó jelentése. Tehát például a beépített egész összeadást nem válthatjuk fel egy olyan összeadó rutinnal, amelyik figyel a túlcsordulásra. A nyelv beépített adattípusaira és azoknak a `*` vagy `[ ]` vagy `&` típusmódosítókkal származtatott típusaira nem definiálhatók újabb műveletek, így például a + operátor értelmezése egész tömbök összeadására nem terjeszthető ki. Az *operator overloading* csak osztályként (**struct** vagy **class**) definiált típusok esetében alkalmazható. Ezt a szabályt az a kényszer is szülte, hogy egy operátor-függvénynek legalább egy **class** típusú argumentuma kell legyen. (A fenti példában az egész értékadás esetében tagfüggvényről van szó, és mint

ismeretes, minden tagfüggvénynek létezik egy, a **this** pointer által megcímzett implicit argumentuma.)

Az operátoroknak a C++ nyelv eddigi definíciójában rögzített precedenciája (kiértékelési sorrendje) és szintaxisa nem változtatható meg. A szóban forgó osztálydefiníciótól és a kérdéses operátor-függvények megvalósításától függetlenül az

```
x == y + z;
```

kifejezés-utasításban először az **operator**+, majd ezt követően az **operator**== függvény kerül végrehajtásra. A szabványos operátorokhoz hasonlóan, a precedencia zárójelezéssel felülbírálható. Az operátorok argumentumszámát az *overloading* kapcsán nem változtathatjuk meg. Négy szabványos operátor (+ - * &) az értelmezés kiterjesztése során mind egyoperandusú, mind kétoperandusú is lehet. A ++ és -- operátorok egyoperandusúak, de nem lehet különbséget tenni a postfix és prefix változatuk között. Mint arra már utaltunk, egy operátor-függvény lehet akár tagfüggvény, akár "barát" függvény, de ugyanazt a műveletet csak egyféleképpen definiálhatjuk. Például halmazok unióját (egyesítését) az összeadás operátorához (kétoperandusú +) rendelhetjük:

```
class halmaz {
            friend halmaz &operator+(halmaz &a, halmaz &b);
            ...
        };
```

illetve

```
class halmaz {
            ...
            public:
             halmaz &operator+(halmaz &a);
            ...
        }
```

Figyeljük meg, hogy amikor tagfüggvényként definiáltuk az összeadás műveletét, akkor eggyel kevesebb argumentumot adtunk meg explicit módon. Ennek az a magyarázata, hogy egy tagfüggvény első operandusa mindig az az implicit osztály-objektum, amelyik az operátor-függvényt aktivizálta. Például az értékadás művelete esetén a

```
b = a;
```

utasítás ekvivalens a

```
b.operator=(a);
```

függvényhívással, ahol az operátor-függvényen belül a **this** pointer *b*-re fog mutatni. Az összeadás műveleténél a tagfüggvényként történő megvalósítás esetleg hátrányos is lehet, mert például a **this**-szel megcímzett első operandust elronthatjuk, ami nyilván nem kívánt mellékhatás. Újból hangsúlyozzuk, hogy egy adott operátor-függvényt vagy csak tagfüggvényként, vagy csak **friend**-ként adhatunk meg, különben a fordítóprogram számára nem lenne egyértelmű, hogy melyik definíciót kell használnia.

Vannak olyan operátorok, amelyekhez tartozó operátor-függvény *csak* tagfüggvényként definiálható. Ezek az operátorok = (értékadás), a `[]` (indexelő operátor), a `()` (aktivizáló operátor) és a `->` (pointeres mezőkiválasztó operátor). Egy tagfüggvényként definiált operátor baloldali operandusa mindig a saját osztályába kell tartozzon. Ha egy operátor baloldali operandusa idegen típusú, akkor a kérdéses operátor-függvény csak **friend** függvény lehet. Így például egy halmaz elemeinek a szokásos << operátorral való kiírása a *cout* szabványos kimeneti adatfolyamba a következőképpen oldható meg:

```
#include <iostream.h>
class halmaz {   ...
         friend ostream& operator<<(ostream &os, halmaz &s);
         ...
       };
...
ostream& operator<<(ostream &os, halmaz &s)
{ int i;
  if (!s.darab)                    // Üres halmaz
  {
    os << "Üres.";
    return os;
  }

  // Az elemeket egy ciklusban kiírjuk:

  for (i = 0; i < s.darab; os << s.elemek[i++] << ' ')
  ;

  return os;
}
```

3.6. Mintapéldák műveleti jelek értelmezésének kiterjesztésére

3.6.1. A halmaz típus műveletei

Célunk a továbbiakban az, hogy a *halmaz* típusunkhoz olyan operátor-függvényeket hozzunk létre, amelyek megvalósítják a különféle halmazalgebrai és relációs műveleteket, például a Pascal programozási nyelvben megszokottakhoz hasonlóan. Tűzzük ki célul a következő műveletek megvalósítását:

Jel	Művelet	Eredmény
+	Halmazok uniója (egyesítése) `halmaz a, b; ... a+b;`	Az egyesített halmaz. A közös elemek csak egyszer forduljanak elő az eredmény halmazban!

+	Halmaz és egy egész szám uniója.
	`halmaz a; int i;`
	`a+i; illetve i+a;`
		A halmazba bevesszük az egész számot is és az így kapott halmazt szolgáltatja eredményül. (Kétféle változat az operandusok sorrendje szerint)

*	Halmazok metszete (közös része) `halmaz a, b; ... a*b;`	A közös rész, tehát azon elemek halmaza, amelyek mindkét operandusban megtalálhatók.

=	Értékadás.
	`halmaz a, b; int i;`
	`a=b; illetve a=i;`
		Balérték halmaz felveszi a jobbérték halmaz vagy egy egész kifejezéssel meghatározott egyelemű halmaz értékét.

<=	Részhalmaz-vizsgálat.
	`int i; halmaz a, b;`
	`i = (a <= b);`
		Igaz értéket ad, ha *a* részhalmaza *b*-nek.

>=	Részhalmaz-vizsgálat.
	`int i; halmaz a, b;`
	`i = (b >= a);`
		Igaz értéket ad, ha *a* részhalmaza *b*-nek.

<	Eleme reláció vizsgálata (Pascal `IN` művelet).
	`int i,j; halmaz a;`
	`j = i < a;`
		Igaz értéket ad, ha i eleme a-nak.

<<	Adatfolyamba való kiíratás.
	`halmaz a; ostream os;`
	`os << a << "valami";`
		Visszaadja az adatfolyamot, hogy láncolható legyen a kiíratás.

Célszerű, ha a megvalósítás során egymásra támaszkodnak az egyes operátor-függvények. Például az összeadás műveletét először terjesszük ki a *halmaz*+*halmaz* esetre. Ezután a *halmaz*+**int** jellegű "összeadást" oly módon kezeljük, hogy a kérdéses operátor-függvény az összeadás jobb oldalán álló egyetlen egész számot egy egyelemű egész tömbbe írja, ezzel aktivizálja a

halmaz::halmaz() konstruktort és az így létrehozott átmeneti halmazváltozót adja hozzá a baloldali halmazhoz a már korábban megírt *halmaz+halmaz* jellegű összeadást megvalósító operátor-függvény segítségével. Ugyanígy járunk el az **int**+*halmaz* típusú összeadásnál, de most a bal oldali operandus kerül egy átmeneti, halmaz típusú munkaváltozóba. Így tehát a *halmaz*::**operator**+() függvénynek több változata lesz, közülük mindig a paraméter-szignatúra alapján választja ki a fordító azt, amelyikkel az adott összeadás megvalósítható.

Ha számbavesszük az imént elképzelt műveleteket, a következőképpen célszerű deklarálnunk a *halmaz* típust:

```
#include <iostream.h>
                // A << operátor-függvény ostream paramétere miatt kell

class halmaz
{
      int   *elemek;
      int   darab;
friend halmaz&   operator+(const halmaz &a, const halmaz &b);
friend halmaz&   operator+(const halmaz &a, const int e);
friend halmaz&   operator+(const int e, const halmaz &a);
friend halmaz&   operator*(const halmaz &a, const halmaz &b);
friend int       operator<(const int e, const halmaz &a);
friend int       operator<=(const halmaz &a, const halmaz &b);
friend int       operator>=(const halmaz &a, const halmaz &b);
friend ostream&  operator<<(ostream &os, halmaz &s);
public:
                halmaz(int n=0, int *e=NULL);
                ~halmaz() { if (darab) delete [darab] elemek; }

      halmaz& operator=(const halmaz &s);
      halmaz& operator=(const int i);
      halmaz& operator+=(const halmaz &s);
      int     operator==(const halmaz &s);
      int     operator!=(const halmaz &s);
};
```

Amint arra utaltunk, az értékadás műveletét *csak* tagfüggvénnyel valósíthatjuk meg. Továbbá minden olyan esetben, amikor egy operátorban alkalmazandó balérték kifejezés olyan *halmaz* típusú kifejezés, amelynek értéke a végzendő művelet tulajdonságai folytán biztosan nem változik meg (== != összehasonlítások), illetve amikor kifejezetten a balérték megváltozása a cél (+=), szintén tagfüggvényként valósítjuk meg az operátor-függvényt. Ha egy kétoperandusú művelet egyik operandusa nem *halmaz* típusú, vagy a bal oldali operandust nem a **this**-en keresztül szeretnénk elérni (hogy értéke ne változhas-

son meg), az operátor-függvényt a *halmaz* osztályon kívüli **friend** függvénnyel valósítjuk meg (+ * < <= >= <<).

Annak érdekében, hogy bizonyos halmazműveleteket egyszerűbben lehessen megvalósítani, az eddigi osztálydefiníciót célszerű egy **private** hozzáférésű tagfüggvénnyel bővítenünk, amely minden esetben, amikor egy halmaz elemeiben változás áll be, a halmaz elemeit nagyság szerint (újra)rendezi. Ezt a rendezési műveletet az *stdlib.h* fejléc állományban deklarált *qsort()* szabványos könyvtári függvény felhasználásával egyszerűen megvalósíthatjuk. Legyen ezen tagfüggvény neve *halmazrendezo*. Az elmondottak alapján az osztálydefiníció a következőképpen néz ki:

```
#include <stdlib.h>
                // A qsort() miatt kell
#include <iostream.h>
                // A << operátor-függvény ostream paramétere miatt kell
class halmaz
{
        int      *elemek;
        int      darab;
friend halmaz&   operator+(const halmaz &a, const halmaz &b);
friend halmaz&   operator+(const halmaz &a, const int e);
friend halmaz&   operator+(const int e, const halmaz &a);
friend halmaz&   operator*(const halmaz &a, const halmaz &b);
friend int       operator<(const int e, const halmaz &a);
friend int       operator<=(const halmaz &a, const halmaz &b);
friend int       operator>=(const halmaz &a, const halmaz &b);
friend ostream&  operator<<(ostream &os, halmaz &s);
public:
                halmaz(int n=0, int *e=NULL);
                ~halmaz() { if (darab) delete [darab] elemek; }

        halmaz& operator=(const halmaz &s);
        halmaz& operator=(const int i);
        halmaz& operator+=(const halmaz &s);
        int     operator==(const halmaz &s);
        int     operator!=(const halmaz &s);
private:
        void    halmazrendezo();

};
```

A halmazrendezést végző tagfüggvény a következő:

```
static int cmp(const void *a,const void *b) // Elrejtjük, ezért static
// Paraméterezés: ahogy qsort() igényli.
// A halmaz::halmazrendezo() összehasonlító függvénye qsort()-hoz
{
  return -1*((*((int*)a)) < (*((int*)b)))
       + 1*((*((int*)a)) > (*((int*)b)));

// Visszatérési érték: qsort() igényei szerinti.
}

void halmaz::halmazrendezo()
{
  qsort(elemek, darab, sizeof(int), cmp);
}
```

Ezt a tagfüggvényt a konstruktorban is aktivizálni kell:

```
halmaz::halmaz(int n, int *e)
{
  int i;

  if (n)       // Ha nem üres halmaz, helyet foglalunk az elemek-nek
  {            // és e[] tömbből n darab számot a halmazba másolunk.
   for (i = 0, darab = n, elemek = new int[n];
        i < n;
        i++)
   {
     elemek[i] = e[i];
   }
   halmazrendezo();
               // A halmazba másolt számokat sorba rendezzük.
  }
  else
  {            // Üres halmazt hozunk létre.
   darab = 0;
   elemek = NULL;
  }
}
```

Az operátor-függvények megvalósításának áttekintését kezdjük a kétféle értékadás művelettel. Ugyan már láttunk példát az értékadásra, de abból kimaradt egy vizsgálat: ha a bal oldali halmaz nem üres, akkor a jobbérték halmaz bemásolása előtt fel kell szabadítanunk a korábbi elemek által lefoglalt memóriát. További kiegészítésként a művelet befejezésekor elvégezzük az elemek sorba rendezését:

```
halmaz& halmaz::operator=(const halmaz &s)
{
  int i;

  if(darab) delete [darab] elemek;// Korábbi elemek[] eldobása.

  darab = s.darab;                // Számosság másolása.
  if (!s.darab) {elemek = NULL;}  // Az új halmaz üres halmaz.
  else                            // Az új halmaz nem üres, így
  {                               // elemek[]-nek helyet kell
    elemek = new int[darab];      // foglalni, utána másolunk.

    for (i = 0; i < darab; i++) elemek[i] = s.elemek[i];
  }
  halmazrendezo();                // A biztonság kedvéért rendezünk
  return *this;                   // *this lesz az értékadó kifejezés
}                                 // értéke.
```

Az értékadás *kötelezően* tagfüggvény. A **this** implicit pointer azt a balérték halmazt címzi meg, aminek értéket szeretnénk adni, még a fenti operátor-függvény *s* paramétere az értékadás jobb oldalán álló halmaz kifejezés lesz. Semmi más dolgunk nincs, mint (a bal oldali halmaz elemei által korábban foglalt memória felszabadítása után) az *s.elemek[ ]* tömböt a *this->elemek[ ]* tömbbe másolni. Ez utóbbit a **new** operátorral hozzuk létre, az *s.elemek[ ]* tömbbel egyező mérettel. Végül a másolás után az elemeket rendezzük a *halmazrendezo()* tagfüggvénnyel, hogy mindig, amikor értékadás történik, a rendezettség biztosított legyen, ugyanis erre a későbbiekben számítani fogunk. Az értékadás művelete visszatérési értékül a balérték kifejezést kell adja, ha igazodni szeretnénk a C, illetve C++ beépített operátorainál megszokottakhoz, így operátor-függvényünket a **return *this;** utasítással zárjuk. (Emlékeztetőül: A C nyelvben az értékadás műveletének *fő hatása* az, hogy az értékadás kifejezés értéke megegyezik az értékadás jobbérték kifejezésének értékével, *mellékhatása* pedig az, hogy a balérték kifejezésként adott változóba bemásolódik a jobbérték kifejezés tartalma. Fenti operátor-függvényünk is ennek megfelelően működik.)

Az értékadás másik formája az, amikor egyetlen egy egész számból álló, egyelemű halmazt hozunk létre:

```
halmaz& halmaz::operator=(const int i)
{
  if (darab) delete [darab] elemek;
  darab = 1;
  elemek = new int[darab];
  elemek[0] = i;
  return *this;
}
```

A megoldás emlékeztet a konstruktorra, de itt explicit módon megadjuk a számosságot (*darab = 1*) és az egyetlen halmazelemet. Mivel csak egyelemű halmaz jön létre, rendezni nem kell.

A következő operátorunk az 'eleme-e' reláció vizsgálata, melynek működése a Pascal nyelv *IN* operátorának felel meg. A művelet jobb oldali operandusa egy halmaz, bal oldali operandusa pedig egy egész szám. A reláció igaz, ha a kérdéses egész szám eleme az adott halmaznak, egyébként hamis. Ezt a műveletet a < jelhez rendeljük, amely emlékeztet a matematikában használatos ∈ jelre ('eleme' reláció jele).

```
int operator<(const int e, const halmaz &a) // Eleme-e? vizsgálat
{
  int i;

  for (i = 0; i < a.darab; i++)
  {
    if (e == a.elemek[i]) return 1;
  }
  return 0;
}
```

A megvalósítás igen egyszerű: amint az *e* egész számot megtaláljuk az *a* halmazban, IGAZ (1) értékkel visszatérhetünk. Ha a kereső ciklusban nem találjuk meg az egész számot a halmazban, akkor HAMIS (0) értékkel térünk vissza.
Tényleges halmazalgebrai művelet az *unióképzés*, amelyet a + operátorhoz rendelünk. A feladat igen egyszerű: a két összeadandó halmaz elemeit egy közös tömbbe kell másolnunk. A halmazalgebra szabályainak betartására azonban ügyelnünk kell: az unióban résztvevő mindkét halmazban előforduló *azonos* elemek az egyesített halmazban csak *egyszer* szerepelhetnek, így a közös halmazba való másoláskor ezt figyelembe kell vennünk. Az egyesített halmaz számossága sem egyszerűen csak a két halmaz számosságának az összege, hiszen a két halmazban szereplő azonos elemeket csak egyszer kell számbavenni. A megvalósítás során már támaszkodunk az imént definiált 'eleme-e' relációs operátorra és a halmaz értékadás műveletére.

```
halmaz& operator+(const halmaz &a, const halmaz &b) // Unióképzés
{
  int i, j, m;                      // Munkaváltozók
  int n = a.darab + b.darab;        // Unió halmaz maximális számossága
  int l = n;
  int *celemek;                     // Unió halmaz elemeit rakjuk ide
  halmaz &ret = *(new halmaz);      // ret az unió halmaz referenciája
```

```
  if (n)                    // Ha nem két üres halmazt adunk össze:
  {
    celemek = new int[n]; // Maximális számossággal helyet foglalunk
    for(i = 0; i < a.darab; i++)
    // Egyik halmaz elemeit mind bemásoljuk
      celemek[i] = a.elemek[i];

// A másik halmazból csak azokat az elemeket másoljuk be, amelyek az
// az első halmazban nem szerepeltek; közben számláljuk a
// másolásokat:

    for (j = 0, m = 0; j < b.darab; j++)
      if (!(b.elemek[j] < a)) celemek[i+ m++] = b.elemek[j];
       // Itt az 'eleme-e' operátorral vizsgáljuk a tartalmazást.

// Első halmaz számossága + a másodikból másolt elemek számossága adja
// a végső számosságot:

    n = a.darab + m;
 // n és celemek[] segítségével létrhozzok az az eredmény halmazt.
    halmaz c(n,celemek);
    delete [l] celemek;           // celemek[] tömböt már eldobhatjuk.
 // A c halmazt a halmazértékadás segítségével a dinamikusan lefoglalt
 // ret halmaznak adjuk értékül. Ez lesz a visszatérési érték.
 // A c halmaz a visszatérés után megszűnik, ret megmarad.
    ret = c;
  }
  return ret;
}
```

Érdekes kérdés az unióhalmaz visszatérési értékként történő átadása. Az unióképzés eredményeként született elemek a *celemek[]* egész tömbbe kerülnek, az eredményhalmaz számossága pedig az *n* változóban lesz. Ezekből a *halmaz* típus konstruktorával képezhetünk egy tényleges *halmaz*-t. Ezt egy, az operátor-függvényen belüli lokális *halmaz* típusú változó definiálásával – a *halmaz c(n,celemek)* utasítással – tehetjük meg. A gond az, hogy a függvényből való visszatéréskor a verem memóriában létrehozott *c* halmaz megszűnik, így azt, vagy az arra vonatkozó referenciát nem adhatjuk át visszatérési értékként. A megoldás az, hogy a **new** operátorral létrehozunk egy üres halmazt a szabad memóriában és az imént definiált, és az unióképzés eredményével feltöltött *c* halmazt ennek értékül adjuk. Az így létrehozott halmaz már nem pusztul el a függvényből való visszatérés után sem, így az erre vonatkozó referencia visszatérési érték lehet. Az unióhalmaz elemeinek sorbarendezését az értékadás operátor-függvénye fogja végrehajtani. Megjegyezzük, hogy a referencia típusú visszatérési értékre azért van szükség, hogy tetszőlegesen láncolt, zárójelezett halmaz-kifejezések is korrekt módon kiértékelhetők legyenek.

A + operátorhoz még egy további értelmezést rendelhetünk: egy halmaz elemkészletének bővítését egyetlen egy egész számmal. Célszerű, ha ez a fajta "összeadás" nem függ az operandusok sorrendjétől, azaz létezik *halmaz*+**int** és **int**+*halmaz* változata is. Ez C++ terminológiával kifejezve azt jelenti, hogy az **operator+**() függvénynek a meglévőn kívül még két további, különböző paraméterszignatúrájú változatát kell elkészítenünk. A megvalósítás során a már meglévő operátor-függvényekre támaszkodunk. Az összeadás *halmaz*+**int** szignatúrájú változatát az unióképzésre vezetjük vissza úgy, hogy az egész számból egy egyelemű halmazt képzünk és alkalmazzuk az unió műveletet:

```
 halmaz operator+(const halmaz &a, const int e)
{
  int *el = new int;// 1 elemű tömböt foglalunk

  *el = e;
  halmaz b(1,el);     // e-t egy egyelemű halmazba tesszük
  delete el;

  return a+b;         // visszavezetjük halmazösszeadásra
}
```

Az **int**+*halmaz* szignatúrájú változatot pedig az imént elkészítettre vezetjük vissza, az operandusok sorrendjének megcserélésével:

```
halmaz operator+(const int e, const halmaz &a)
{
  return a+e; // int+halmaz-t visszavezetjük halmaz+int-re
}
```

Az eddigiek alapján oldjuk meg a következő feladatot!

Készítsük el a += operátor kiterjesztését a *halmaz* típusra! Működése egyezzen meg a szabványos += operátorával: a balértékként adott halmazhoz adjuk hozzá a jobbértékként szereplő halmaz-kifejezést!

Megoldás:

Tekintve, hogy a += operátor esetében a bal oldali operandust (ami balérték kifejezés, azaz változó kell legyen) módosítani szeretnénk, illetve valójában értékadási műveletet kell végeznünk, az operátor-függvényt tagfüggvényként valósítjuk meg. Tehát egy *a*+=*s* típusú halmazművelet végrehajtása egyenlő a

this**=this**+*s* végrehajtásával, ahol **this** az a halmazra mutat. Ezzel meg is adtuk a megoldás vázát. A konkrét megvalósítás például a következő lehet:

```
/* halmaz.cpp-ben található */
halmaz& halmaz::operator+=(const halmaz &s)
{
  int i;
  halmaz a(darab,elemek); // a halmaz tartalma azonos lesz *this-szel

  a = a+s;                // Elvégezzük az összeadást a+= s értelemben

// Eldobjuk *this régi tartalmát majd *this-t a értékével töltjük fel:

  delete [darab] elemek;  // Régi this->elemek-et felszabadítjuk

  darab = a.darab;        // Új this->darab
  elemek = new int[darab];// Új this->elemek
  for (i = 0; i < darab; i++) elemek[i] = a.elemek[i];

  return *this;
}
```

Írjunk operátor-függvényt a metszet (közös rész) képzés megvalósítására! A műveletet rendeljük a * operátorhoz!

Megoldás:

Meggondolásaink hasonlók kell legyenek, mint amiket az unió művelet kapcsán tettünk. Éppen ezért az unióhoz hasonlóan **friend**-ként célszerű megvalósítanunk az operátor-függvényt. További megfontolásaink már csak a halmazalgebrával kapcsolatosak. A közös rész halmaz a két kiinduló halmazból csak azokat az elemeket tartalmazza, amelyek mindkettőben megtalálhatók. Célszerű tehát úgy előállítani két halmaz metszetét, hogy kezdetben üres halmaznak tekintjük a közös részt, majd például sorra vesszük a kisebbik számosságú halmaz elemeit és megvizsgáljuk, hogy megtalálhatóak-e a másik halmazban. (Erre felhasználhatjuk az $\in$ reláció vizsgálatára korábban megírt **operator**< ( ) operátor-függvényünket.) Ha egy elem a másik halmazban is megtalálható, felvesszük az eredményhalmazba. A konkrét megvalósítás a következő lehet:

```
/* halmaz.cpp-ben található */
halmaz& operator*(const halmaz &a, const halmaz &b)
{
  int i,                        // Ciklusváltozó
      m,                        // Kisebb számosságú halmaz elemszáma
      n,                        // Nagyobb számosságú halmaz elemszáma
      l;
  int *celemek;                 // A metszet halmaz elemeihez
  halmaz nagyobb, kisebb;

 // Eldöntjük, hogy a két operandus közül melyik a nagyobb számosságú:

  if (a.darab > b.darab)
  {
    n = a.darab;
    m = b.darab;
    nagyobb = a;
    kisebb = b;
  }
  else
  {
    n = b.darab;
    m = a.darab;
    nagyobb = b;
    kisebb = a;
  }

  // A közös rész halmaz nem lehet nagyobb, mint a nagyobbik halmaz,
  // így annak elemszámával foglalunk helyet celemek[] tömb számára.

  celemek = new int[n];
  l = n;
  n = 0; // n-nel azt számláljuk majd, hogy hány közös elemet találunk
  // A kisebbik halmaz minden elemére megvizsgáljuk, hogy az eleme-e a
  // nagyobbik halmaznak. Ha igen, a közös rész halmazba kerül:

  for (i = 0; i < m; i++)
  {
    if (kisebb.elemek[i] < nagyobb)
    {
      celemek[n++] = kisebb.elemek[i];
    }
  }
  halmaz c(n,celemek), // c-ben a közös rész
  &ret = *(new halmaz);
                       // Csak new-val allokált halmazt adhatunk vissza.
  delete [l] celemek;
  ret = c;             // Eredmény másolása a visszatérési értékbe.
  return ret;
}
```

Készítsünk operátor-függvényeket a részhalmaz reláció vizsgálatára a <=, illetve a >= operátorok értelmezésének kiterjesztésével! Az `a<=b` reláció legyen IGAZ akkor, ha az `a` halmaz részhalmaza `b`-nek!

Megoldás:

Logikai (**int**) típusú lesz a visszatérési érték, ezért **friend** függvényt írunk. A működés könnyen átgondolható:

1. Üres halmaz minden halmaznak részhalmaza, ezért ilyen esetben a visszatérési érték IGAZ kell legyen.
2. Nagyobb számosságú halmaz nem lehet kisebb számosságú halmaz részhalmaza, ezért ilyen esetben a visszatérési érték HAMIS kell legyen.
3. Az $A \subseteq B$ reláció (A részhalmaza B-nek) csak akkor igaz, ha A halmaz minden eleme eleme B halmaznak, azaz minden $a \in A$-ra igaz, hogy $a \in B$.

Komolyabb programozási munkát csak a 3. pont kidolgozása igényel, de ez is egy egyszerű **for** ciklussal és az 'eleme-e' relációs operátor felhasználásával könnyen megoldható:

```
/* halmaz.cpp */
int operator<=(const halmaz &a, const halmaz &b)
{
  int i, ret;

  // Üres halmaz minden halmaznak részhalmaza:
 if (!a.darab) return 1;

  // Nagyobb számosságú halmaz nem lehet kisebb számosságú
  // halmaz részhalmaza:

  if (a.darab > b.darab) return 0;

  // ret csak akkor lesz 1, ha a valódi részhalmaza b-nek:

  for (i = 0, ret = 1; i < a.darab; i++)
     ret &= (a.elemek[i] < b);

  return ret;
}
```

A fordított reláció vizsgálatát egy egyszerű operandus cserével a fenti függvényre vezetjük vissza:

```
int operator>=(const halmaz &a, const halmaz &b)
{
  return b <= a; // Operanduscserével visszavezetjük <= műveletre.
}
```

Készítsünk operátor-függvényeket két halmaz egyenlőségének, ill. különbözőségének a vizsgálatára!

```
/* halmaz.cpp-ben található */
int halmaz::operator==(const halmaz &s)
{
  int i;

  if (darab != s.darab) return 0;   // Különböző számosságú halmazok.
  if (!darab) return 1;              // Két üres halmaz.

  // Két valódi, azonos számosságú halmaz:

  for (i = 0; i < darab; i++) if (elemek[i] != s.elemek[i]) return 0;

  return 1;
}

int halmaz::operator!=(const halmaz &s)
{
  return !(s == *this);
}
```

A fenti megoldásokhoz magyarázatként csak annyit érdemes hozzáfűzni, hogy az ‘egyenlő-e’ reláció vizsgálatakor arra támaszkodunk, hogy

– a halmazaink rendezettek, és
– nem tartalmaznak ismétlődő elemeket.

Készítsünk egy keretprogramot, amellyel kipróbálhatjuk eddigi operátorainkat!

Megoldás:

```
int aelemek[] = {3,4,5,6};
int main(void)
{
  int i;
  halmaz a(sizeof(aelemek)/sizeof(int), aelemek);
  halmaz b,c,d;

  cout << "A 'halmaz' típus és operátorainak kipróbálása.\n"
       << "============================================\n\n";
```

```
    cout << "A 'main'-ben definiált halmazok: A, B, C és D.\n";
    cout << "Ezek tartalma:\n";
    cout <<  "A halmaz - " << a << " B halmaz - " << b <<
            " C halmaz - " << c << " D halmaz - " << d << '\n';

    cout << "Az unióképzés és értékadás művelete (halmaz+int)"
            " az A = A+1 példával:\n";
    a = a+1;
    cout << "A halmaz - " << a << '\n';

    cout << "Összetett uniókifejezés B = ((B+2)+7)+3 \n";
    b = ((b+2)+7)+3;
    cout << "B halmaz - " << b << '\n' ;

    cout << "A halmaz=int értékadás a C = 13 példával: ";
    c = 13;
    cout << "C halmaz - " << c << '\n';

    cout << "Az unió művelet (halmaz+halmaz) a B += A utasítással: \n";
    b += a;
    cout << "B halmaz - " << b << '\n';

    cout << "Sokszoros értékadás és halmaz+int művelet:\n";
    d = c = c+4;
    cout << "A D = C = C+4 eredménye D-ben: " << d <<
         " és C-ben: " << c << '\n';

    if (d == c) cout << "D és C halmaz megegyezik!\n";
    else        cout << "D és C halmaz különbözik!\n";

    cout << "Közös rész képzés a D = C * A művelettel: ";
    d = c*a;
    cout << "D halmaz - " << d << '\n';

    if (d <= a) cout << "A D halmaz részhalmaza az A halmaznak.\n";

    cout << "Az eleme-e reláció vizsgálata: '4' eleme-e D-nek?\n";
    if (4 < d) cout << "Igen, a '4'-es szám eleme a D halmaznak.\n";

    return 0;
}
```

A teljes példaprogram HALMAZ.CPP néven megtalálható a lemezmellékleten. Ezt Borland C++ rendszerben lefordítva, a futási eredmény a következő lesz a DOS-ban:

```
A 'halmaz' típus és operátorainak kipróbálása.
==============================================

A 'main'-ben definiált halmazok: A, B, C és D.
Ezek tartalma:
A halmaz - 3 4 5 6  B halmaz - Üres. C halmaz - Üres. D halmaz - Üres.
Az unióképzés és értékadás művelete (halmaz+int) az A = A+1 példával:
A halmaz - 1 3 4 5 6
Összetett uniókifejezés B = ((B+2)+7)+3
B halmaz - 2 3 7
A halmaz=int értékadás a C = 13 példával: C halmaz - 13
Az unió művelet (halmaz+halmaz) a B += A utasítással:
B halmaz - 1 2 3 4 5 6 7
Sokszoros értékadás és halmaz+int művelet:
A D = C = C+4 eredménye D-ben: 4 13  és C-ben: 4 13
D és C halmaz megegyezik!
Közös rész képzés a D = C * A művelettel: D halmaz - 4
A D halmaz részhalmaza az A halmaznak.
Az eleme-e reláció vizsgálata: '4' eleme-e D-nek?
Igen, a '4'-es szám eleme a D halmaznak.
```

Gyakorlatok:

1. Terjesszük ki a halmazokra a *= operátort, ahol a * művelet a metszetképzést jelenti!

2. Készítsünk operátor-függvényt halmazok *különbségének* képzésre!

3. Paraméterezett osztály (*template*) használatával alakítsuk át úgy a programot, hogy más alaptípusból is képezhessünk halmazokat!

Ezt a ***main***-t és eddigi programrészleteinket oly módon készítettük el, hogy nem használtunk ki semmi, a Borland C++ rendszerhez kötődő sajátságot. Így programunk *hordozható,* azaz más géptípuson, más operációs rendszer alatt, más C++ fordítóval is fordítható, illetve futtatható. Programunkat kipróbáltuk egy DEC 3000/300 XL Alpha processzoros munkaállomáson OpenVMS/AXP operációs rendszer alatt, a Digital C++ fordítóval lefordítva, illetve egy SUN SparcUltra munkaállomáson, Solaris operációs rendszer alatt, a SUN C++ fordítójával lefordítva. A futási eredményeket az alábbi képernyőmásolatok szemléltetik:

```
DECterm 1
File  Edit  Commands  Options  Print                                  Help
ATH_POPPE> cxx halmaz
ATH_POPPE> link halmaz
ATH_POPPE> run halmaz
A 'halmaz' típus és operátorainak kipróbálása.
==================================================

A 'main'-ben definiált halmazok: A, B, C és D.
Ezek tartalma:
A halmaz - 3 4 5 6  B halmaz - Üres. C halmaz - Üres. D halmaz - Üres.
Az unióképzés és értékadás müvelete (halmaz+int) az A = A+1 példával:
A halmaz - 1 3 4 5 6
Összetett uniókifejezés B = ((B+2)+7)+3
B halmaz - 2 3 7
A halmaz=int értékadás a C = 13 példával: C halmaz - 13
Az unió mûvelet (halmaz+halmaz) a B += A utasítással:
B halmaz - 1 2 3 4 5 6 7
Sokszoros értékadás és halmaz+int mûvelet:
A D = C = C+4 eredménye D-ben: 4 13  és C-ben: 4 13
D és C halmaz megegyezik!
Közös rész képzés a D = C * A mûvelettel: D halmaz - 4
A D halmaz részhalmaza az A halmaznak.
Az eleme-e reláció vizsgálata: '4' eleme-e D-nek?
Igen, a '4'-es szám eleme a D halmaznak.
ATH_POPPE>
```

Futási eredmény OpenVMS/AXP operációs rendszerben

```
xterm
liszt>
liszt> CC halmaz.cxx
liszt> a.out
A 'halmaz' típus és operátorainak kipróbálása.
==================================================

A 'main'-ben definiált halmazok: A, B, C és D.
Ezek tartalma:
A halmaz - 3 4 5 6  B halmaz - Üres. C halmaz - Üres. D halmaz - Üres.
Az unióképzés és értékadás müvelete (halmaz+int) az A = A+1 példával:
A halmaz - 1 3 4 5 6
Összetett uniókifejezés B = ((B+2)+7)+3
B halmaz - 2 3 7
A halmaz=int értékadás a C = 13 példával: C halmaz - 13
Az unió mûvelet (halmaz+halmaz) a B += A utasítással:
B halmaz - 1 2 3 4 5 6 7
Sokszoros értékadás és halmaz+int mûvelet:
A D = C = C+4 eredménye D-ben: 4 13  és C-ben: 4 13
D és C halmaz megegyezik!
Közös rész képzés a D = C * A mûvelettel: D halmaz - 4
A D halmaz részhalmaza az A halmaznak.
Az eleme-e reláció vizsgálata: '4' eleme-e D-nek?
Igen, a '4'-es szám eleme a D halmaznak.
liszt>
```

Futási eredmény Solaris operációs rendszerben

3.6.2. A sztring típus és a hozzátartozó műveletek definiálása

További mondanivalónk illusztrálására egy újabb típust definiálunk: a *sztring*-et, amely karaktertömbök sztringszerű kezelését teszi lehetővé. Így például megengedi a tetszőleges hosszúságú karakterfüzérek használatát, biztosítja az indexelést, a sztringkapcsolást, valamint a sztring-összehasonlítást. Bár ezen funkciók az ANSI C szabványos könyvtári függvényei révén rendelkezésre állnak, mégis érdemes ezeket a C++ nyelv lehetőségeinek felhasználásával megvalósítanunk, hiszen a sztringműveleteket végző programok gyors fejlesztéséhez a BASIC vagy Turbo Pascal programozási nyelvekben megszokott funkciók nagy segítséget nyújthatnak.

Alapvetően az *operator overloading* lehetőségével fogunk élni. A korábbi programpéldánk (halmazok) kapcsán bemutatott lehetőségekhez képest újdonságként bemutatjuk az *indexelőoperátor*, a *függvényaktivizáló operátor*, valamint a *típusmódosító operátor* értelmezésének kiterjesztését az új sztring típusunkra, valamint példát mutatunk az ún. *másoló konstruktor (copy constructor)* definiálására. A következőtáblázatban összefoglaljuk a sztring típusra kiterjesztendő operátorokat.

Jel	Mûvelet	Eredmény
+	Sztringek összefûzése (konkatenálása) `sztring a, b; ... a+b;`	Az egyesített sztring, amelynek elsõ részét az `a`, második részét a `b` sztring alkotja.
+	Sztring és karaktertömb (karaktersztring konstans) összefûzése (konkatenálása) `sztring a; ... a+"valami"`	Az egyesített karakterlánc, amelynek elsõ részét az `a` sztring, második részét pedig a karaktertömb-konstans alkotja.

=	Értékadás sztringnek.
	`sztring a, b; char c[]="xx";`
	`a=b;` illetve `a=c;`
		Balérték sztring felveszi a jobbérték sztring (kifejezés) vagy a `char[]` konstans értékét.

`[]`	Sztring indexelése.
	`sztring a("alma"); int i=1;`
	`char ch = a[i];`
		A sztring `i`-edik karakterét adja vissza.

`()`	Aktivizáló operátor.
	`sztring a("alma");`
	`char ch = a();`
		Minden aktivizáláskor a sztring soronkövetkezõelemét adja vissza, ha a sztring végére ért, kezdi elölrôl. (ún. iterátor)

`(char*)`	Típuskonverziós operátor
	`sztring a("alma"); char *p;`
	`p = (char*)(a);`
		A sztringet `char*` típussá (karaktertömbre mutató pointerré) konvertálja.

`==`	Egyenlôség vizsgálata.
	`int i; sztring a, b;`
	`i = (a == b);`
		Igaz értéket ad, ha az `a` sztring azonos a `b`-vel.

`!=`	Egyenlõtlenség vizsgálata.
	`int i; sztring a, b;`
	`i = (b != a);`
		Igaz értéket ad, ha az `a` sztring különbözik a `b`-tôl.

Jel	Mûvelet	Eredmény

<	Kisebb reláció vizsgálata.
	`int i; sztring a,b;`
	`  j = (a < b);`
		Igaz értéket ad, ha a kisebb, mint b, azaz az ABC-ben elôbb szerepel.

>	Nagyobb reláció vizsgálata.
	`int i; sztring a,b;`
	`  j = (a > b);`
		Igaz értéket ad, ha b kisebb, mint a, azaz az ABC-ben elôbb szerepel.

<<	Output adatfolyamba való kiíratás.
	`sztring a; ostream os;`
	`  os << a << "valami";`
		Kiírja a sztringet és visszaadja az adatfolyamot, hogy láncolható legyen a kiíratás.

>>	Olvasás input adatfolyamból.
	`sztring a; istream is;`
	`  is >> a;`
		Egy egyszavas sztringet olvas be a bemenő steram-ből. Visszaadja az adatfolyamot, hogy láncolható legyen a beolvasás.

Ezek után nézzük a vonatkozó osztálydefiníciót:

```
/* sztring.cpp-ben található */
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <iostream.h>

class sztring {
  private:
     char *szoveg;                          // sztring értéke
     int  hossz;                            // sztring hossza
     int  index;                            // aktuális elem

  public:
     // Konstruktorok

     sztring(char *s = NULL);               // Tömbbel hozza létre
     sztring(sztring &s);                   // Copy konstruktor

     // Destruktor (in-line):

     ~sztring()
       { if (szoveg != NULL) delete [hossz+1] szoveg; }

     // Tagfüggvények:

     int  hossza(void) { return hossz; }
     void nagybetusit(void);
     void kisbetusit(void);

     // Értékadó operátorok:

     sztring& operator=(sztring& s);
                    // sztring = sztring értékadás
     sztring& operator=(char* p);
                    // sztring = char[] értékadás
```

```
    // Sztring konkatenáló operátorok:

    sztring& operator+(sztring &s);
                    // sztring+sztring konkatenálás
    sztring& operator+(char* p);
                    // sztring+char[] konkatenálás

    // Relációs operátorok

    int operator==(sztring &s){ return !strcmp(szoveg,s.szoveg); }
    int operator!=(sztring &s){ return !strcmp(szoveg,s.szoveg); }
    int operator<(sztring &s) { return 0 > strcmp(szoveg,s.szoveg); }
    int operator>(sztring &s) { return 0 < strcmp(szoveg,s.szoveg); }

    // Indexelőoperátor

    char&   operator[](const int ind);
                    // Indexelőop., kötelezően tag-fv.

    // "Következőeleme" (ierátor) operátor

    char    operator()();
                    // Kötelezően argumentum nélküli

    // Típuskonverziós operátor

    operator char*();
                    // Kötelezően argumentum és típus nélküli

private:
    // Indexhatár-ellenőrzés: privát függvénnyel

    int hatar_ok(int ind);    // Igaz, ha ind index lehet.

// Sztring kiiratása, beolvasása

friend ostream& operator<<(ostream &os, sztring &s);
                    // Kiiratás stream-be
friend istream& operator>>(istream &is, sztring &s);
                    // Beolvasás stream-ből

};
```

Sztring típusunk három privát hozzáférésű adatmezőt tartalmaz: a tárolt szöveg hosszát (egész szám), a tárolt szöveget (karakterpointer) és egy harmadik egész számot, amely az iterátor függvény által használt index.

A sztring osztály számos tagfüggvényt tartalmaz, ezek többsége a fenti táblázatnak megfelelőoperátor-függvény. Ezek, akárcsak a konstruktorok,

publikus elérésű függvények. Az adatfolyamokkal kapcsolatos operátor-függvények, akárcsak a korábbi halmaz osztály esetében, itt is **friend** függvények.

Másoló konstruktor

Részletes tárgyalásunkat kezdjük a konstruktorokkal! Amint az az osztálydeklarációból kiderül, kétféle konstruktort készítettünk. Az egyik **char[]** tömbökből (pl. hagyományos C sztringkonstansokból) hoz létre sztring objektumot. Az osztálydeklarációban a konstruktor karaktertömbre mutató pointer paraméteréhez a NULL értéket rendeljük alapértelmezésül. Ha tehát ennek folytán a sztring-osztály aktuálisan létrehozandó objektum-példányában a *szoveg* mező NULL értéket kap (azaz ha paraméter nélkül deklarálunk egy sztring objektumot) akkor a sztring karakterei számára nem foglalunk memóriát. Ez azt jelenti, hogy a paraméter nélkül deklarált sztring objektumok automatikusan üres sztringek lesznek. Ha egy karaktertömbbel aktivizáljuk ezt a konstruktort, akkor a ***strlen()*** könyvtári függvény segítségével maghatározzuk a tömb hosszát, az értékes karaktereknek és a karakterfüzér végét jelző '\0' karakternek helyet foglalunk, majd a sztring objektumot az inicializálásra használt karaktertömbbel töltjük fel.

```
/* sztring.cpp-ben található */
sztring::sztring(char *s)
// Karaktertömbből sztringet készítőkonstruktor
{
   if (s != NULL)
   {
      hossz = strlen(s);
      szoveg = new char[hossz+1]; // +1 hely EOS-nak
      index = 0;
      strcpy(szoveg, s);
   }
   else
   {
      hossz = 0; szoveg = NULL; index = 0;
   }
}
```

Hasznos lehet az, amikor egy új objektumpéldányt egy már létező objektumpéldány másolataként hozunk létre. Az ilyen konstruktort másoló konstruktornak (*copy constructor*) szokás nevezni. A sztring osztály esetében is másik konstruktorunk ilyen:

```
/* sztring.cpp-ben található */
sztring::sztring(sztring &s)
// Sztringből sztringet készítő(copy) konstruktor
{
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
  else
  {
     hossz = 0; szoveg = NULL; index = 0;
  }
}
```

A destruktor dolga pusztán a sztring karakterei által foglalt memóriaterület felszabadítása, éppen ezért ezt **inline** formában valósítottuk meg (lásd az osztálydeklarációban).

A sztring típus operátor-függvényei

Az értékadó operátorok megvalósítása nagyon hasonló a konstruktorokhoz. Példaként tekintsük a sztring-sztring értékadást:

```
/* sztring.cpp-ben található */
sztring& sztring::operator=(sztring& s)
// Ugyanolyan, mint a copy konstruktor
{
  if (hossz) delete [hossz+1] szoveg; // Méret elhagyható.
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
  else
  {
     hossz = 0; szoveg = NULL; index = 0;
  }
  return *this;
}
```

A másoló konstruktorhoz képest itt annyi a különbség, hogy a **this** által mutatott balérték kifejezés korábbi értékét el kell dobni, illetőleg a sztring

másolás művelet után ***this**-t visszatérési értékként szolgáltatjuk (hogy láncolható legyen az sztring értékadás művelete).

A konkatenálás műveletét a + operátorhoz rendeljük. A halmaz típus kapcsán bemutatott megoldástól eltérően itt most tagfüggvényként valósítjuk meg a vonatkozó operátor-függvényeket. Az alábbiakban a sztring+sztring műveletet megvalósító operátor-függvényt mutatjuk be:

```
/* sztring.cpp-ben található */
sztring& sztring::operator+(sztring& s)
// sztring+sztring konkatenálás
{
  char *kozos_szoveg = NULL;
  int kozos_hossz = hossz + s.hossz;
  sztring *ret = new sztring;

  if (kozos_hossz)
  {
     kozos_szoveg = new char[kozos_hossz + 1];
     kozos_szoveg[0] = '\0';
     strcat(kozos_szoveg,szoveg);
     strcat(kozos_szoveg,s.szoveg);
  }

  *ret = kozos_szoveg; // sztring = char* értékadás

  if (kozos_hossz) delete [kozos_hossz + 1] kozos_szoveg;
                        // A tömbméret megadás elhagyható
  return *ret;

}
```

Az összeadás művelet bal oldali operandusát az tagfüggvényként definiált operátor-függvény implicit paramétere (***this**) szolgáltatja, a jobb oldali operandus pedig a függvény `s` paramétere lesz.

A két sztring együttes hosszának megfelelő memóriaterület lefoglalása után a tényleges összefűzést az ***strcat()*** könyvtári függvény segítségével végezzük el. A visszatérési értéket a halmaz típus kapcsán leírt meggondolások alapján, egy **new** operátorral létrehozott sztring objektum referenciájaként szolgáltatjuk.

Ez eddig leírtak alapján fejlesszük tovább a sztring típus operátor-függvényeit lépésről lépésre!

Készítsük el a sztring összefűzés (konkatenálás) művelet sztring+karaktertömb szignatúrájú változatát!

A megoldás során támaszkodjunk az eddigi analógiákra!

A megoldás a lemezmellékleten a SZTRING.CPP file-ban megtalálható.

Készítsünk operátor-függvényeket a sztringekre értelmezett relációs műveletekhez!

Megoldás:

```
#include <string.h>

class sztring {
  private:
    char *szoveg;                            // sztring értéke
    int  hossz;                              // sztring hossza
    int  index;                              // aktuális elem
  public:
    ...
    int operator==(sztring &s){ return !strcmp(szoveg,s.szoveg); }
    int operator!=(sztring &s){ return !strcmp(szoveg,s.szoveg); }
    int operator<(sztring &s) { return 0 > strcmp(szoveg,s.szoveg); }
    int operator>(sztring &s) { return 0 < strcmp(szoveg,s.szoveg); }
    ...
};
```

A relációs műveletek operátor-függvényeit az ***strcmp()*** szabványos könyvtári függvény "megfejelésével", készítettük el. Tekintve, hogy ezek az operátor-függvények nagyon rövidek, azokat **inline** tagfüggvényként (azaz bináris makróként) definiáltuk az osztálydefiniciós blokkban. Ezen operátor-függvények dolga tehát a könyvtári karaktertömb-összehasonlító függvény visszatérési értékének átkódolása a relációs műveleteknél szokásos logikai értékekre.

A megoldás a lemezmellékleten a SZTRING.CPP file-ban megtalálható.

Folytassuk a sztringkezelés operátor-függvényeinek az elkészítését!

Terjesszük ki a << operátor értelmezését sztringek kimeneti adatfolyamba való kiírására!

A megoldás során induljunk ki a halmaz típusnál alkalmazott operátor-függvényből, azt alakítsuk át! Jelen esetben is "barátként" definiáljuk az operátor függvényt, amely első argumentumként egy *ostream&* típusú adatot vár (ebbe a *stream*-be fogunk írni) és visszatérési értékként is ezt az adatfolyamot adja (miután bele írtunk), hogy láncolható legyen a kiiratás. A korábbi megoldáshoz képest annyi a módosítás, hogy a sztring *szoveg* mezőjét közvetlenül is kiirathatjuk az adatfolyamba (a << operátor **char*** típusra értelmezett változatával), míg a halmaz típus esetén a halmaz elemeit egy ciklussal tudtuk csak kiiratni. Íme, a megoldás:

```
/* sztring.cpp-ben található */
ostream& operator<<(ostream &os, sztring &s) // Kiiratás stream-be
{
   if (s.hossz) os << s.szoveg;
   return os;
}
```

A fenti operátor-függvény mintájára készítsük el a >> operátor sztringeket beolvasó változatát is!

Gondolatmenetünk hasonló az operátor-függvény paramétereit és visszatérési értékét illetően, mint a beolvasás esetében. Egy karaktertömbbe, mint bufferbe közvetlenül is beolvashatunk egy teljes szót. Ezt aztán a *sztring*=**char*** típusú ártákadással a sztring-referencia típusú argumentumnak értékül adhatjuk:

```
/* sztring.cpp-ben található */
istream& operator>>(istream &is, sztring &s) // Beolvasás stream-ből
{
  char buffer[256];

  is >> buffer;
  s = buffer;
  return is;
}
```

Miután a halmaz típus analógiájára elkészítettük az alapvető sztringműveleteket (értékadás, összekapcsolás, kiiratás, beolvasás, relációs műveletek) végző operátor-függvényeinket, készítsük el azon további

függvényeinket, amelyek a karaktertömbökkel való kompatibilitáshoz kellenek. Ezek közül is a legfontosabb az *indexelő* operátor.

Indexelő operátor

Írjunk operátor-függvényt a sztring típusú adatok indexeléséhez!

Megírandó operátor-függvényünk kötelezően *tagfüggvény* kell legyen. Visszatérési értékének típusa `char&`, így egy indexelt sztring tetszőleges karakterpozíciójába *írhatunk* is. Annak kivédésére, hogy hibás indexeléssel se történjék súlyos futási hiba, készítettünk egy privát hozzáférésű tagfüggvényt (***hatar_ok()*** függvény), amellyel az indexelt sztring aktuális indexhatárai ellenőrizhetők. Indexhatár túllépésnél (az egyszerűsége kedvéért) a szabványos hiba adatkimenetre egy hibaüzenetet írunk és visszatérési értékként a sztring kezdőkarakterét szolgáltatjuk. Helyes indexelés esetén a sztring értékét tároló karaktertömb megfelelő elemét szolgáltatja a függvény:

```
/* sztring.cpp-ben található */
char& sztring::operator[](const int ind)
// Indexelőop., kötelezően tag-fv.
{
  if (hatar_ok(ind)) return szoveg[ind];
  else
  {
    cerr << "\nHibás indexelés: az "
         << ind
         << " index kívül esik a határokon.\n";
    return szoveg[0];
  }
}
```

Saját típuskonverziós operátor készítése

Az indexelhetőségen túl fontos feladat a karaktertömbökkel való kompatibilitás szempontjából a típuskonverzió. A **char*** $\Rightarrow$ *sztring* konverzió a megfelelő konstruktor, illetve értékadó operátor révén megoldott, de a sztringekre mutató karakterpointerek előállításához saját típuskonverziós operátort kell írnunk.

Készítsünk saját típoskonverziós operátort sztringeknek karaktertömbre mutató pointerré való konvertálásához!

Az elkészítendő operátot-függvény szintén *kötelezően* tagfüggvény kell legyen, és *nem rendelkezhet* semmilyen paraméterrel.

```
sztring::operator char*() // Típuskonverziós operátor
{
   return szoveg;
}
```

A típuskonverzió operátor-függvényének az azonosítója meg kell egyezzen azzal a típussal, amibe konvertálni szeretnénk; feni példánkban ez **char***. Operátor-függvényünknek semmi más dolga nincs, mint a sztring értékét tároló tömb kezdőcímét szolgáltatni. Egyszerűsége révén ez a függvény akár inline függvény is lehetne.

Iteráló operátor

Kényelmi szolgáltatásként definiáljunk még egy operátor-függvényt, amely minden egyes aktivizálás alkalmával egy adott sztring következő karakterét adja vissza! Ezt a funkciót rendeljük az aktivizáló operátorhoz!

Az ilyen operátort *iteráló operátornak* szokás nevezni. Megvalósítása a sztring típusunk esetében következő:

```
char sztring::operator()()  // Kötelezően argumentum nélküli
{
  if (index < hossz) return szoveg[index++];
     else return szoveg[index = 0];
}
```

A függvényaktivizáló () operátorhoz rendelt operátor-függvény *kötelezően argumentum nélküli*. Látható, hogy az az aoperátor-függvény, amelynek a kedvéért az `index` adattagot definiáltuk. Ha az indexeléssel a sztring végére értük, kezdjük elölről.

Az iterátor függvény használatát az alábbi programpélda szemlélteti:

```
sztring d(" Ebből a sztringből kiszedjük a ., és \" írásjeleket");

cout << "Írásjelek kiirtása az indexelőés függvényaktivizáló"
           " operátorokkal:\n";
```

```
for (int i = 0; i < d.hossza(); i++)
{
  char ch;

  ch = d(); // d sztring következőkaraktere
  switch (ch)
  {
    case PONT:
    case VESSZO:
    case IDEZO:
               d[i] = ' ';
               break;
  }
}
cout << "Írásjelektől megfosztott D sztring:\n" << d << '\n';
```

A sztring típus további taggfüggvényei

Kényelmi szolgáltatásként még három tagfüggvényt definiáltunk (lásd a lemezmellékleten a SZTRING.CPP állományt). Az elsővel egy sztring hossza kérdezhető le, ezt használtuk a fenti mintaprogramban is. A másik két publikus hozzáférésű tagfüggvénnyel egy sztring karaktereit vagy kisbetűssé, vagy nagybetűssé alakíthatjuk. Megvalósításukhoz a *ctype.h* fejléc állományban deklarált ***toupper()*** és ***tolower()*** rutinokat használjuk fel.

Írjunk egy kipróbáló keretprogramot a sztring típus használatának demonstrálására!

Megoldás:

```
// sztring.ccp-ben található
#include <stdio.h>

enum {PONT = '.', VESSZO = ',', IDEZO = '\"'};
int main(void)
{
  sztring a("Ez az \"A\" sztring, ");
  sztring b("ez pedig a \"B\" sztring. ");

  sztring c;
  sztring d;
  sztring e(a);

  cout << "\n\nSaját 'sztring' típus lehetőségeinek bemutatása:\n\n";

  cout << "sztring(char*) konstruktorral létrehozott sztringek:\n";
  cout << a << '\n';
```

```
   cout << a << '\n';
   cout << b << '\n';
   cout << "sztring(sztring) copy konstruktorral "
           "létrehozott sztring:\n";
   cout << e << " - az A sztringből hoztuk létre.\n";
   cout << "sztring+sztring konkatenálás művelete: C = A+B. "
           "Íme, a C sztring:\n";
   c = e + b;
   cout << c << '\n';
   cout << "sztring+char* konkatenálás művelete: D = C+\"konstans\"."
           "Íme:\n";
   d = c + "Meg egy hozzáadott sztring konstans.\n";
   cout << d << '\n';

   cout << "Írásjelek kiirtása az indexelőés függvényaktivizáló"
            " operátorokkal:\n";
   for (int i = 0; i < d.hossza(); i++)
   {
     char ch;

     ch = d(); // d sztring következőkaraktere
     switch (ch)
     {
        case PONT:
        case VESSZO:
        case IDEZO:
                  d[i] = ' ';
                  break;
     }
   }
   cout << "Írásjelektől megfosztott D sztring:\n" << d << '\n';

   cout << "Tagfüggvények bemutatása.\nAdj meg egy sztringet: ";
   cin >> e;
   e.kisbetusit();
   cout << "Ez csupa kisbetűvel: " << e << '\n';
   e.nagybetusit();
   cout << "Ez csupa nagybetűvel: " << e << '\n';

   cout << "\nTípuskonverzió bemutatása printf(\"%s\")-sel:\n";
   printf("%s\n",(char*)(a));

   a = "aaa";
   b = "bbb";

   if (a < b) cout << "\nAz " << a << " sztring kisebb, mint a "
                      << b << " sztring.";
   return 0;
}
```

A program futásának az eredménye a képernyőn:

```
sztring(char*) konstruktorral létrehozott sztringek:
Ez az "A" sztring,
ez pedig a "B" sztring.
sztring(sztring) copy konstruktorral létrehozott sztring:
Ez az "A" sztring,  - az A sztringből hoztuk létre.
sztring+sztring konkatenálás művelete: C = A+B. Íme, a C sztring:
Ez az "A" sztring, ez pedig a "B" sztring.
sztring+char* konkatenálás művelete: D = C+"konstans". Íme:
Ez az "A" sztring, ez pedig a "B" sztring. Meg egy hozzáadott sztring
konstans.

Írásjelek kiirtása az indexelőés függvényaktivizáló operátorokkal:
Írásjelektől megfosztott D sztring:
Ez az  A  sztring  ez pedig a  B  sztring  Meg egy hozzáadott sztring
konstans

Tagfüggvények bemutatása.
Adj meg egy sztringet: Alma
Ez csupa kisbetűvel: alma
Ez csupa nagybetűvel: ALMA

Típuskonverzió bemutatása printf("%s")-sel:
Ez az "A" sztring,

Az aaa sztring kisebb, mint a bbb sztring.
```

Vegyük észre, hogy a sztring típus ***kisbetusit()*** és ***nagybetusit()*** tagfüggvényei, illetve operátor-függvényeinek egy része (pl. a relációs műveletek) pusztán egyszerüsített írásmódot jelentenek a hagyományos C könyvtári függvények meghívása helyett. Ezek használatával csínján kell bánnunk, mert ugyan C++ forráskódunk sokkal elegánsabban néz ki a kiterjesztett jelentésű operátorok alkalmazása révén, de a futtatható programkód hatékonysága rosszabb, mintha mi magunk "kézzel" kódolnánk le az egyes műveleteket. Egyszerű példáinknál (egész számok halmaza, sztringek) ez talán még nem jelentős veszteség, de gondoljunk csak arra, hogy ha pl. mátrixalgebrai műveletekre is kiterjesztjük a standard C operátorok jelentését, akkor egy egyszerű aritmetikai műveleti jel is igen nagyszámú elemi műveletet takarhat (pl. osztás-jel — mátrix-invertálás).

Végül, a teljesség kedvéért közöljük a sztring típus teljes definicióját a kipróbáló keretprogrammal együtt:

```
#include <stdlib.h>
#include <sztring.h>
#include <ctype.h>
#include <iostream.h>

class sztring {
  private:
     char *szoveg;                  // sztring értéke
     int  hossz;                    // sztring hossza
     int  index;                    // aktuális elem

  public:
     // Konstruktorok

     sztring(char *s = NULL);       // Tömbbel hozza létre
     sztring(sztring &s);           // Copy konstruktor

     // Destruktor (in-line):

     ~sztring()
       { if (szoveg != NULL) delete [hossz+1] szoveg; }
                                    //  ^^^^^^^karakterszám elhagyható
     // Tagfüggvények:

     int  hossza(void) { return hossz; }
     void nagybetusit(void);
     void kisbetusit(void);

     // Értékadó operátorok:

     sztring& operator=(sztring& s);
                                    // sztring = sztring értékadás
     sztring& operator=(char* p);
                                    // sztring = char[] értékadás

     // Sztring konkatenáló operátorok:

     sztring& operator+(sztring &s);
                                    // sztring+sztring konkatenálás
     sztring& operator+(char* p);
                                    // sztring+char[] konkatenálás
     // Relációs operátorok

     int operator==(sztring &s){ return !strcmp(szoveg,s.szoveg); }
     int operator!=(sztring &s){ return !strcmp(szoveg,s.szoveg); }
     int operator<(sztring &s) { return 0 > strcmp(szoveg,s.szoveg); }
     int operator>(sztring &s) { return 0 < strcmp(szoveg,s.szoveg); }

     // Indexelőoperátor

     char&   operator[](const int ind);
                                    // Indexelőop., kötelezően tag-fv.
```

```
    // "Következőeleme" (ierátor) operátor

    char     operator()();
                                        // Kötelezően argumentum nélküli

    // Típuskonverziós operátor

    operator char*();
                                   // Kötelezően argumentum és típus nélküli

private:
    // Indexhatár-ellenőrzés: privát függvénnyel

    int hatar_ok(int ind);       // Igaz, ha ind index lehet.

// Sztring kiiratása, beolvasása

friend ostream& operator<<(ostream &os, sztring &s);
                                        // Kiiratás stream-be
friend istream& operator>>(istream &is, sztring &s);
                                        // Beolvasás stream-ből

};
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring::sztring(char *s)  // Tömbből sztringet készítőkonstruktor
{
   if (s != NULL)
   {
      hossz = strlen(s);
      szoveg = new char[hossz+1]; // +1 hely EOS-nak
      index = 0;
      strcpy(szoveg, s);
   }
   else
   {
      hossz = 0; szoveg = NULL; index = 0;
   }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring::sztring(sztring  &s)  //  Sztringből  sztringet  készítő(copy)
konstruktor
{
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
```

```
  else
  {
     hossz = 0; szoveg = NULL; index = 0;
  }
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring& sztring::operator=(sztring& s) // Ugyanolyan, mint a copy
konstruktor
{
  if (hossz) delete [hossz+1] szoveg;
  if (s.hossz)
  {
     hossz = s.hossz;
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, s.szoveg);
  }
  else
  {
     hossz = 0; szoveg = NULL; index = 0;
  }
  return *this;
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring& sztring::operator=(char *p) // Ugyanolyan, mint a tömb
konstruktor
{
  if (hossz) delete [hossz+1] szoveg;
  if (p != NULL)
  {
     hossz = strlen(p);
     szoveg = new char[hossz+1]; // +1 hely EOS-nak
     index = 0;
     strcpy(szoveg, p);
  }
  else
  {
     hossz = 0; szoveg = NULL; index = 0;
  }
  return *this;
}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring& sztring::operator+(sztring& s) // sztring+sztring
konkatenálás
{
  char *kozos_szoveg = NULL;
  int kozos_hossz = hossz + s.hossz;
  sztring *ret = new sztring;
```

```
  if (kozos_hossz)
  {
    kozos_szoveg = new char[kozos_hossz + 1];
    kozos_szoveg[0] = '\0';
    strcat(kozos_szoveg,szoveg);
    strcat(kozos_szoveg,s.szoveg);
  }

  *ret = kozos_szoveg; // sztring = char* értékadás

  if (kozos_hossz) delete [kozos_hossz + 1] kozos_szoveg;

  return *ret;

}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring& sztring::operator+(char* p) // sztring+char[] konkatenálás
{
  char *kozos_szoveg = NULL;
  int kozos_hossz = hossz + strlen(p);
  sztring *ret = new sztring;

  if (kozos_hossz)
  {
    kozos_szoveg = new char[kozos_hossz + 1];
    kozos_szoveg[0] = '\0';
    strcat(kozos_szoveg,szoveg);
    strcat(kozos_szoveg,p);
  }

  *ret = kozos_szoveg; // sztring = char* értékadás

  return *ret;

}
//+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
char& sztring::operator[](const int ind)
// Indexelőop., kötelezően tag-fv.
{
  if (hatar_ok(ind)) return szoveg[ind];
  else
  {
    cerr << "\nHibás indexelés: az "
         << ind
         << " index kívül esik a határokon.\n";
    return szoveg[0];
  }
}
```

```
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
int sztring::hatar_ok(int ind)
{
   return (0 < ind) && (ind < hossz);
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
char sztring::operator()()  // Kötelezően argumentum nélküli
{
  if (index < hossz) return szoveg[index++]; else index = 0;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++
ostream& operator<<(ostream &os, sztring &s) // Kiiratás stream-be
{
   if (s.hossz) os << s.szoveg;
   return os;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
istream& operator>>(istream &is, sztring &s) // Beolvasás stream-ből
{
  char buffer[256];

  is >> buffer;
  s = buffer;
  return is;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
sztring::operator char*() // Típuskonverziós operátor
{
   return szoveg;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void sztring::kisbetusit(void)
{
  for (int i = 0; i < hossz; szoveg[i++] = tolower((*this)()));
  index = 0;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
void sztring::nagybetusit(void)
{
  for (int i = 0; i < hossz; szoveg[i++] = toupper((*this)()));
  index = 0;
}
//++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

// Vége a sztring típus definíciójának.
// =================================
```

```
// SZTRING.CPP-ben található

#include <stdio.h>

enum {PONT = '.', VESSZO = ',', IDEZO = '\"'};
int main(void)
{
  sztring a("Ez az \"A\" sztring, ");
  sztring b("ez pedig a \"B\" sztring. ");

  sztring c;
  sztring d;
  sztring e(a);

  cout << "\n\nSaját 'sztring' típus lehetőségeinek bemutatása:\n\n";

  cout << "sztring(char*) konstruktorral létrehozott sztringek:\n";
  cout << a << '\n';
  cout << b << '\n';
  cout << "sztring(sztring) copy konstruktorral "
          "létrehozott sztring:\n";
  cout << e << " - az A sztringből hoztuk létre.\n";
  cout << "sztring+sztring konkatenálás művelete: C = A+B. "
          "Íme, a C sztring:\n";
  c = e + b;
  cout << c << '\n';
  cout << "sztring+char* konkatenálás művelete: D = C+\"konstans\"."
          "Íme:\n";
  d = c + "Meg egy hozzáadott sztring konstans.\n";
  cout << d << '\n';

  cout << "Írásjelek kiirtása az indexelőés függvényaktivizáló"
           " operátorokkal:\n";
  for (int i = 0; i < d.hossza(); i++)
  {
    char ch;

    ch = d(); // d sztring következőkaraktere
    switch (ch)
    {
       case PONT:
       case VESSZO:
       case IDEZO:
                 d[i] = ' ';
                 break;
    }
  }
  cout << "Írásjelektől megfosztott D sztring:\n" << d << '\n';

  cout << "Tagfüggvények bemutatása.\nAdj meg egy szót: ";
  cin >> e;
  e.kisbetusit();
```

```
    cout << "Ez csupa kisbetűvel: " << e << '\n';
    e.nagybetusit();
    cout << "Ez csupa nagybetűvel: " << e << '\n';

    cout << "\nTípuskonverzió bemutatása printf(\"%s\")-sel:\n";
    printf("%s\n",(char*)(a));

    a = "aaa";
    b = "bbb";

    if (a < b) cout << "\nAz " << a << " sztring kisebb, mint a "
                    << b << " sztring.";
    return 0;
}
```

Megjegyzés: A Borland C++ 5.02 rendszerben a string.h fejléc fájl beszerkesztésével rendelkezésünkre áll a imént bemutatott *sztring* típushoz hasonló, annál sokkal bővebb szolgáltatásokat nyújtó *string* típus. Használatával kapcsolatban a rendszer kézikönyvére ill. Help lehetőségére utalunk.

3.7. Nagyobb méretű programok futtatása

A nagyobb méretű programok több modulból, azaz több forrásfájlból állnak és csoportmunkánál ezeket esetleg különböző személyek is írják. Ezenkívül általában felhasználásra kerülnek korábban – vagy mások által – készített rutinok lefordított formában, tárgykódú fájlokként vagy könyvtárakba (.lib állományok) gyűjtve. Maga a végső program a - Borland C++ szóhasználattal: a project – tehát nagyszámú fájl kezelését teszi lehetővé A több modul használata nagy programok esetén azért is szükségszerű mert a fordító nem képes akármekkora forrásállományt feldolgozni. Minden modul önállóan is lefordítható. Egy fejléc fájlra több forrásfájl hivatkozhat, azaz egy .h állomány módosítása több .cpp forrásfájl újbóli fordítását is szükségessé teheti.

3.7.1. Project fájl betöltése Borland 3.1 verzió esetén

Project fájl kiterjesztése .prj. Amelyik program .prj fájllal rendelkezik, annak futtatáshoz először a ***Project*** menü ***Open project*** almenüjén kattintva a megjelenő listadobozból a betöltendő .prj fájlt kiválasztjuk, illetve kettős kattintással betöltjük. A project fájl betöltése után a *Project* ablakban megjelenik a project fájl tartalma. Ha valamelyik .CPP fájlt javítani kell, akkor egy kettős rákattintás betölti a szerkesztői ablakba. A ***Run*** hatására futtathatjuk a programot.

3.7.2. Project fájl létrehozása

A ***Project*** menüpont kiválasztásakor megjelenő ***Open project*** ablak *Open project file* szerkesztői ablakába írjuk be a létrehozandó project fájl nevét .prj kiterjesztéssel, majd az OK hatására létrejön egy üres project ablak, fejlécében a létrehozott project fájl neve. Az ***Add item...*** menüpont segítségével vehetünk fel egy újabb modult a project listába. A felvitelnél ugyanazt a forrásfájlt többször nem tudjuk bevinni. Törlésre is van lehetőség. Kijelöljük a törlendő tételt a project ablakban és a ***Delete item...*** menüpont kiválasztásával végrehajtjuk a törlést. A .h fájlt nem kell a project fájlba betölteni.

A ***Compile/Build all*** menüpont kiválasztásával az összes fájl lefordul, míg a ***Make*** csak a változtatott fájlokat fordítja. ***Run*** hatására futtatjuk a lefordított programot.

3.7.3. Műveletvégző feladat több modulban

A virtuális tagfüggvénnyel működő MUV_OWN4.CPP állományban tárolt programot alakítottuk át több modulra. A *muv_fgc.h* fájlban található a műveletvégző osztálydefiníciói. A *muv_fgc.cpp* állomány tartalmazza az osztályok tagfüggvényeit és a *muv_main.cpp* állományban van a főprogram. A *muv_fgc.h* fejléc fájlt mindkét .cpp fájlban #include segítségével beszerkesztettünk. Egynél több .cpp állomány esetén már .PRJ fájlt kell létrehozni a Windows 3.1 alatt működő Borland C++ 3.1. rendszerben.
A P3_07_V3 alkönyvtár tartalmazza a modulokat. DOS rendszerből indítva a fordítót , a fordítás és szerkesztés az alábbi módon történik:

```
bcc muv_main muv_fgc
```

```
/* muv_fgc.h */
#include <iostream.h>
#include <math.h>
class Adatok
{
  public:
    float x,y;
  public:
    Adatok(float x1,float y1);
};
class MuveletiJel
{
  public:
    char muv;
  public:
    MuveletiJel(char muv1);
};
class Eredmeny
{
  public:
     float eredmeny;
  public:
     Eredmeny();
};
class Muvelet:public MuveletiJel,
              public Adatok,
              public Eredmeny
{  public:
             Muvelet(char muv1, float x1, float y1);
             Muvelet();
             ~Muvelet();
             virtual void Szamol();
             void Vegrehajt();
             void Kiir();
 };
```

```
class SajatMuvelet:public Muvelet
{
  public:
    SajatMuvelet(char,float,float);
    ~SajatMuvelet();
    void Szamol();
};

/* muv_fgc.cpp */
#include "muv_fgc.h"
Adatok::Adatok(float x1, float y1)
{
  x = x1;
  y = y1;
}
MuveletiJel::MuveletiJel(char muv1)
{
  muv = muv1;
}

Eredmeny::Eredmeny()
{
  eredmeny = 0;
}

Muvelet::Muvelet(char muv1, float x1, float y1):
         MuveletiJel(muv1),
         Adatok(x1,y1),
         Eredmeny()
{
}
Muvelet::~Muvelet()
{
}
void Muvelet::Szamol()
{
 switch (muv)
 {
      case '+' : eredmeny = x+y; Kiir(); break;
      case '-' : eredmeny = x-y; Kiir(); break;
      case '*' : eredmeny = x*y; Kiir(); break;
      case '/' : eredmeny = x/y; Kiir(); break;
  default: cout << "Hibás műveleti jel! \n";
 }
}
void Muvelet::Vegrehajt()
{
  Szamol();
}
```

```
void Muvelet::Kiir()
{
            cout << x << " " << muv << " " << y << " = "
            << eredmeny << endl;
}

SajatMuvelet::SajatMuvelet(char muv1, float x1, float y1):
             Muvelet(muv1,x1,y1)
{
}
SajatMuvelet::~SajatMuvelet()
{
}
void SajatMuvelet::Szamol()
{
 switch (muv)
 {
      case '+' : eredmeny = x+y; Kiir(); break;
      case '-' : eredmeny = x-y; Kiir(); break;
      case '*' : eredmeny = x*y; Kiir(); break;
      case '/' : eredmeny = x/y; Kiir(); break;
      case '^' :
            if( x > 0) { eredmeny = pow(x,y); Kiir(); }
            else
            if ( x == 0) { eredmeny = 0; Kiir(); }
            else {  cout << "negativ alap\n";  break; }
 }
}

/* muv_main.cpp */
#include "muv_fgc.h"

void main()
{
 SajatMuvelet *pz;
 char muv1;
 float x1,y1;
      cout << "müvelet (+,-,*,/,^): "; cin >> muv1;
      cout << "1. adat: "; cin >> x1;
      cout << "2. adat: "; cin >> y1;
      pz = new SajatMuvelet(muv1,x1,y1);
      pz->Vegrehajt();
      delete pz;
}
```

A program futásának eredménye:

```
művelet (+,-,*,/,^): ^
1. adat: 2
2. adat: 3
2 ^ 3 = 8
```

3.7.4. Futtatás Borland C++ 5.02 rendszerben

A Windows'95-ből DOS rendszerbe kilépve a P3_07_V5 könyvtárban lévő modulokból szerkesztjük meg a programot az alábbi módon:

```
C:\CPPEL\P3_07_V5>bcc muv_main muv_fgc
Borland C++ 5.2 Copyright (C) 1987, 1997 Borland International
Mar 19 1997 17:29:40
muv_main.cpp:
muv_fgc.cpp:
Turbo Link Version 7.1.32.2. Copyright (C) 1987, 1996 Borland International
```

Futtatás:

```
C:\CPPEL\P3_7_V5>muv_main
művelet (+,-,*,/,^): ^
1. adat: 2
2. adat: 3
2 ^ 3 = 8
```

NYELVOKTATÓ CD-ROM SOROZAT RAJZFILMMEL

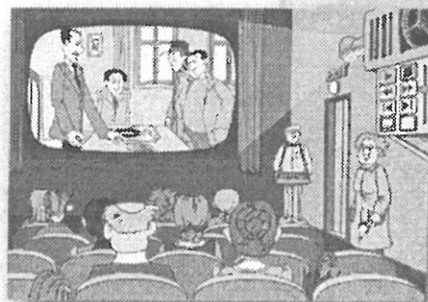
K E Z D Ő / Ú J R A K E Z D Ő

lopva ANGOLUL

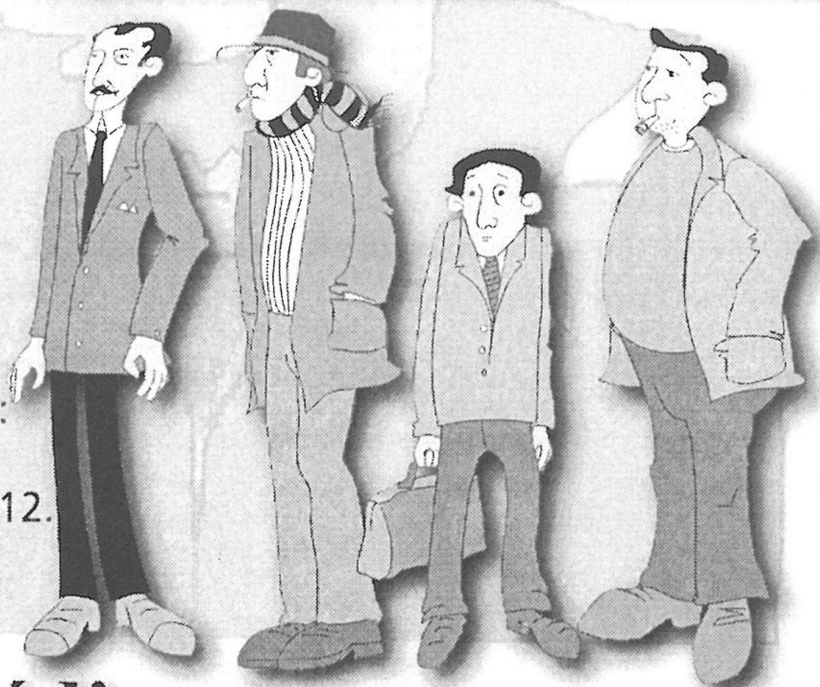
COMPFAIR '97
VÁSÁRDÍJ

HunDidac '97
EZÜSTDÍJ

A 60 perces rajzfilm 12 epizódjához 360 feladat, illetve gyakorlat tartozik: szövegértési, olvasási, írási, szókincset bővítő, nyelvtani tudást fejlesztő gyakorlatok, mikrofonos drillek, valamint játékok.
INTERNETES feladatjavítási lehetőség!
Mindezeket 1500 szavas hangos szótár, kifejezéstár és 50 oldalas, lényegre törő multimédia nyelvtankönyv egészíti ki.
A CD anyaga 80-100 óra alatt dolgozható fel.



KULCS AZ ANGOL NYELVHEZ



ÉRDEKLŐDÉSÜKET
AZ ALÁBBI CÍMEKEN VÁRJUK:
ComputerBooks
1126 Budapest, Tartsay V. u. 12.
Tel.: 1/ 175-1564
info@computerbooks.hu

Profi-Média, a minőségi oktatóprogramok gyártója
6500 Baja, Déri Frigyes sétány 4. • Tel./fax: (36) 79/325-467
www.profi-media.com • pmed@profi-media.com

4. C++ programok szöveges és grafikus üzemmódban

A Borland C++ rendszer - a szabványos ANSI C, illetve C++ könyvtárak mellett - nagyon gazdag rutinkészlettel rendelkezik, melynek segítségével

- DOS, illetve BIOS rutinhívásokat használhatunk,
- magasabb szintű függvények hívásával kezelhetjük az PC képernyőjét szöveges üzemmódban,
- grafikus üzemmódban,
- overlay szervezéssel kihasználhatjuk a PC 640 Kbyte-on felüli memóriáját.

4.1. Szöveg és grafika

A Borland C++ grafikus függvényeinek igen gazdag könyvtára lehetővé teszi, hogy különféle ábrákat, diagramokat rajzoljunk a képernyőre. A PC képernyőjét *text* (szöveges) üzemmódban, vagy *grafikus* üzemmódban használhatjuk. A képernyőt e kétfajta üzemmódban azonban csak felváltva lehet működtetni,

A PC-nek többfajta video adaptere lehet. A monochrom *Display Adapter* (MDA) csak szöveges üzemmódban használható, míg a *Color Graphics Adapter* (CGA), az *Enhanced Graphics Adapter* (EGA), a *Video Graphics Array* (VGA) vagy a *Herculer Monochrome Graphics Adapter* nevű kártyák akár szöveges, akár grafikus módban használhatók. Minden adapternek a szöveges üzemmódban választhatunk, hogy egy sorban 40 vagy 80 oszlop legyen a képernyőn, a sorok száma pedig 25, 43 vagy 50 lehet. Ez utóbbi a képernyőadapter típusától függ. A képernyő üzemmódokat a ***textmode***, ***initgraph*** és a ***setgraphmode*** függvényekkel váltogathatjuk.

- Text üzemmódban a PC képernyője cellákra van osztva (80 vagy 40 oszlop széles és 25, 43 vagy 50 sor magas). Minden cella egy karaktert tartalmaz. Az adott színű és intenzitású karakterek, mint ASCII karakterek jelennek meg a képernyőn. A Borland C++ széles skáláját nyújtja azoknak a függvényeknek, amelyekkel a képernyőre közvetlenül írhatunk különböző színű és intenzitású szövegeket.
- Grafikus üzemmódban a PC képernyője képpontokra (pixel) van osztva. A képpontok száma és színezési lehetősége függ a képernyő adapter típusától és a kiválasztott grafikus üzemmódtól. A Borland C++ grafikus könyvtára

lehetővé teszi, hogy a grafikus képernyőre vonalakat, alakzatokat rajzolhassunk, adott színnel és mintával kitöltött zárt alakzatokat stb.

Text üzemmódban képernyőn a bal felső sarok koordinátája (1,1), az x koordináták balról jobbra, az y koordináták pedig a képernyő tetejétől az aljáig növekednek. Grafikus módban a bal felső sarok koordinátája (0,0), az x és y koordináták értékének növekedése a text üzemmódéhoz hasonló.

A Borland C++ függvényeivel szöveges üzemmódban a képernyőn ablakokat (*window*) hozhatunk létre és a létező ablakokat különbözőképpen manipulálhatjuk is. Az ablak egy téglalap alakú tartomány. Amikor egy alkalmazói program a képernyőre ír például a ***cprintf*** függvénnyel, az eredménye az aktív ablakban jelenik meg. Az ablakok szélén automatikus vágás történik, így az aktív ablakon kívüli képernyő tartományok változatlanok maradnak.

Az alapértelmezés szerinti (default) ablak a teljes képernyő, ezt változtathatjuk kisebbre a ***window*** függvénnyel. Ez a függvény a képernyő koordinátákban megadott pozícióira helyezi el az ablakot.

Grafikus üzemmódban szintén definiálhatunk egy téglalap alakú tartományt a képernyőn. Ez a grafikus ablak, az ún. *viewport*. A grafikus ablak egy virtuális képernyő, amelynek a határain szintén automatikus vágás történik. Amikor tehát rajzolunk a grafikus ablakba, és egyes vonalak az ablak határai túl lógnának, a *viewport* szélein történő vágás miatt a képernyőnek az aktuális *viewport*-on kívül eső részei változatlanok maradnak. A grafikus ablak képernyő koordinátáit a ***setviweport*** függvénnyel állíthatjuk be.

4.1.1. Programozás szöveges üzemmódban

Röviden összefoglaljuk a Borland C++ függvényeket, amelyeket a *text* üzemmódban használhatunk. Ezek a közvetlen konzol I/O függvények (például *cprintf*, *cputs*, stb.), amelyek nagy sebességű szöveg-outputot eredményeznek, a szöveges ablak kezelő függvények, a kurzor pozícionáló rutinok és a szövegattribútum (szín, háttérszín, fényerősség, stb.) kezelő függvények. Ezek prototípusai a conio.h állományban találhatók.

Borland C++ text üzemmódjában a függvények a hat lehetséges mód valamelyikében működhetnek. Az aktuális *text* módot a *textmode* függvény hívásával állíthatjuk be. A *text* mód függvényeit 5 csoportba sorolhatjuk:

- szövegkiírás és -kezelés,
- ablak és üzemmód vezérlés,
- attribútum beállítás,
- állapot lekérdezés,
- kurzor kezelés.

Szöveg írása, olvasása és kezelése

cprintf	formátum szerint ír ki a képernyőre
cputs	sztringet küld a képernyőre
getche	olvas egy karaktert és kiírja a képernyőre

Szöveg és kurzor mozgatása a képernyőn

clrscr	törli a szöveg képernyő ablakot
clreol	törli a sort a kurzor pozíciótól a sor végéig
delline	törli a sort, ahol a kurzor áll.
gotoxy	pozícionálja a kurzort
insline	beszúr egy üres sort azon sor alá, ahol a kurzor áll
movetext	szöveget másol egyik képernyő területről a másikra

Szövegblokk másolása

gettext	szöveget másol egy képernyő területéről a memóriába
puttext	szöveget másol a memóriából egy képernyő területre

A *_wscroll* globális változó vezérli a képernyőre kiírt szöveg görgetését (scroll üzemmód). Ha e változó értéke 1, akkor a szövegkiírás a következő sorba kerül és görgetés történik, ha szükséges. Ha értéke 0, akkor nincs görgetés. A *_wscroll* változó értéke alapértelmezés szerint 1.

Ablak- és az üzemmódvezérlés

textmode	beállítja a képernyőt text módba.
window	definiál egy text módú képernyő ablakot

Attribútum (tulajdonság) beállítás

textattr	egyszerre állítja be az előtér és a háttér színét
textcolor	beállítja az előtér színét
textbackground	beállítja a háttér színét
highvideo	magas fényű intenzitás beállítása
lowvideo	alacsony fényű intenzitás beállítása
normvideo	az eredeti intenzitás beállítása

A szövegattribútum tárolása 8 biten történik. Egy karakterpozícióhoz rendelt attribútum byte alsó 4 bitje az előtér színét tárolja, a következő három biten a háttér színe, a legmagasabb helyiértékű biten pedig a villogást (BLINK) engedélyező flag található.

Állapotlekérdezés

gettextinfo	feltölti a *textinfo* struktúrát az aktuális text módú ablak információival
wherex	megadja a kurzor helyzetének x koordinátáját
wherey	megadja a kurzor helyzetének y koordinátáját

A ***gettextinfo*** függvény feltölti a text.info struktúrát (amely a conio.h fájlban van deklarálva) az alábbi információkkal:
- az aktuális video mód
- az ablak helyzete abszolút képernyő koordinátákban
- az aktuális előtér és háttér színe
- a kurzor aktuális pozíciója

Lekérdezhetjük a kurzor az ablak bal felső sarkához viszonyított pozícióját a ***wherex*** és a ***wherey*** függvényekkel. A ***_setcursortype*** függvény segítségével meg tudjuk változtatni a kurzor alakját. A függvény bemenő paramétere határozza meg a kurzor új alakját.

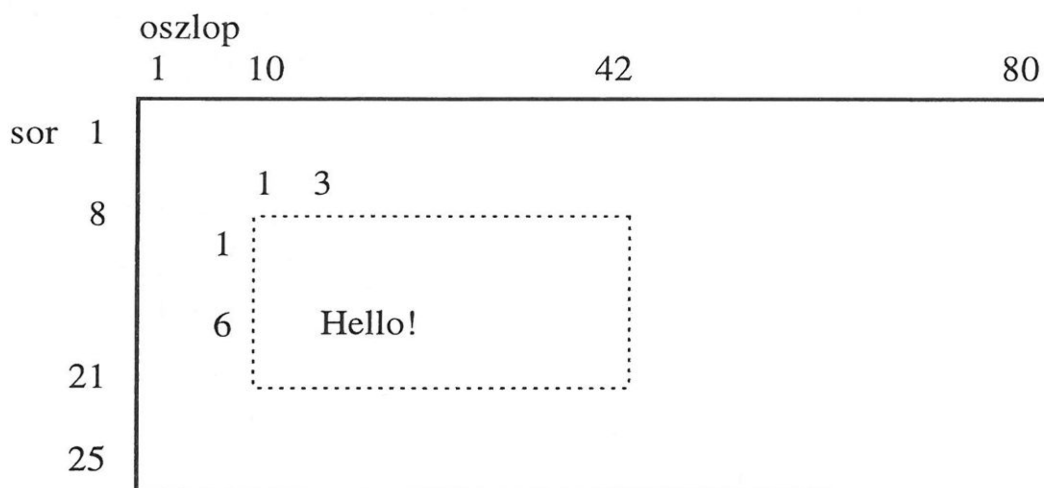
E paraméter értéke háromféle lehet:

_NOCURSOR	eltünteti a kurzort
_SOLIDCURSOR	tömör téglalap alakú, vagy
_NORMALCURSOR	normál, aláhúzás alakú lesz a kurzor

Ablakok szöveges üzemmódban

Text üzemmódban az alapértelmezés szerinti ablak a teljes képernyő, amely maximálisan 50 sort és 40 vagy 80 oszlopot (üzemmódtól függően) tartalmaz. Az "origó" (1,1) a képernyő ablak bal felső sarka.

Például definiálhatjuk egy 80 x 25 méretű ablakban egy kisebb ablakot, amelyeknek az "origója" (bal felső sarka) a (10,8) koordinátájú karakterpozíción kezdődik, és a "vége" (jobb alsó sarka) a (42,21) koordinátájú pontban van. A Hello! szöveget a kis ablak (1,1) koordinátájú bal felső sarkához képest a (3,6) koordinátájú pozíción kezdve írjuk ki (lásd a 4.1. ábrát).



4.1. ábra Szöveges ablakok a képernyőn

```
window(10,8,42,21);
gotoxy(3,6);
cputs("Hello");
```

Hétfajta *text* módot állíthatunk be a képernyőn a ***textmode*** függvény hívásával. E függvény *mode* paramétere határozza meg a beállítandó szöveges üzemmódot. A mode paraméter *text_mode* típusú. Ez egy felsorolt típus, amely a conio.h állományban van definiálva. A *text_modes* típus lehetővé teszi az

üzemmód kiválasztásánál a szimbolikus névvel történő megadást. A ***textmode*** függvény *mode* paramétere számára definiált szimbólikus értékek a következők:

Szimbólikus konstans	Numerikus érték	Üzemmód
LASTMODE	-1	Az előző text módot állítja
BW40	0	Fekete-fehér, 40 oszlop
C40	1	16 szín, 40 oszlop
BW80	2	Fekete-fehér, 80 oszlop
C80	3	16 szín, 80 oszlop
MONO	7	Monokróm, 80 oszlop
C4350	64	EGA 80x43, VGA 80x50

Az attribútum beállítás során megadható színek szimbólikus nevei és konstansai a következők:

Szín	Szimbólum	Érték	Előtér vagy háttér
fekete	BLACK	0	mindkettő
kék	BLUE	1	mindkettő
zöld	GREEN	2	mindkettő
tűrkiz	CYAN	3	mindkettő
piros	RED	4	mindkettő
lila	MAGENTA	5	mindkettő
barna	BROWN	6	mindkettő
világosszürke	LIGHTGRAY	7	mindkettő
sötétszürke	DARKGRAY	8	csak előtér
világoskék	LIGHTBLUE	9	csak előtér
világoszöld	LIGHTGREEN	10	csak előtér
világostürkiz	LIGHTCYAN	11	csak előtér
világospiros	LIGHTRED	12	csak előtér
világoslila	LIGHTMAGENTA	13	csak előtér
sárga	YELLOW	14	csak előtér
fehér	WHITE	15	csak előtér
villogás	BLINK	128	csak előtér

Az első 8 szín előtér és háttér színként használhatók, a 8-tól 15-ig terjedő színek csak előtér színként használhatók. A szöveg villogtatása esetén a BLINK konstanst a színhez kell adni. Például:

```
textcolor(RED+BLINK);
```

4.1.2. Programozás grafikus üzemmódban

A Borland C++ grafikus könyvtára több, mint 70 grafikus függvényt tartalmaz, a magas szintű függvényektől (ilyen a ***setviewport***, ***bar3d*** és a ***drawpoly***) a bit-orientált függvényekig (mint például a ***getimage*** és a ***putimage***).

Az eljárások között van pont, vonal, kör, körív, ellipszis, ellipszisív, téglalap, poligon és téglatest rajzoló, sőt vannak olyan rutinok, amelyek segítségével különböző színnel és mintával befesthetők az alakzatok. A rajzolásra felhasznált vonal vastagsága és mintája is változtatható.

Az ábrákon elhelyezendő szövegek megjelenítésére többfajta karakterkészlet áll rendelkezésre. A kiírandó szöveg mérete változtatható és az írás iránya is beállítható.

A rajzolásnál használt aktuális pointer (*current Pointer* = CP) hasonló a text üzemmód kurzorjához, eltekintve attól, hogy ez nem látszik a képernyőn. Definiálhatunk olyan képernyőablakot, amely levágja az ábra ablakból kieső részeit, azonban ez a vágás nem vonatkozik a CP-re, az a *viewport*-on kívülre is helyeződhet.

A grafikus függvények a *graphics.lib* run-time könyvtárában vannak, a deklarációk és konstansok a *graphics.h* állományban találhatók. A grafikus csomag használatához szükség van a megfelelő grafikus meghajtókra (*.bgi* állományok) és a felhasznált karakter készleteket tartalmazó *.chr* állományokra. A grafikus függvényeket az alábbi módon használhatjuk.

Egy grafikát használó alkalmazói program fordításához és futtatásához a forrásprogram mellett az alábbi állományok is szükségesek.
- graphics.lib könyvtár,
- a megfelelő grafikus meghajtó (*.bgi*)
- karakterkészlet használata esetén a megfelelő font állomány (*.chr* állomány)

A Borland C++ grafikus függvényeit az alábbiak szerint csoportosíthatjuk:
- grafikus rendszer vezérlése,
- rajzolás és festés,
- képernyő- és ablakkezelés,
- szöveg kiírása képernyőre,
- szín beállítás,
- hibakezelés,
- állapot lekérdezése.

A grafikus rendszer vezérlése

closegraph	lezárja a grafikus rendszert.
detectgraph	megvizsgálja a hardvert, eldönti, hogy milyen grafikus rendszert használjunk, és ajánl egy grafikus üzemmódot.
graphdefaults	az alapértelmezés szerint beállítja a grafikus rendszer változóit.
graphfreemem	felszabadítja a grafikus memóriát.
graphgetmem	lefoglalja a grafikus memóriát.
getgraphmode	visszatér az aktuális grafikus móddal.
getmodrange	visszatér a specifikált meghajtó legkisebb és a legnagyobb felbontásával.
initgraph	inicializálja a grafikus rendszert, a hardvert grafikus módba helyezi.
installuserdriver	installálja a felhasználó által írt grafikus meghajtókat.
installuserfont	betölti a felhasználó által készített karakter készletet a BGI karakter állomány táblájába.
restorecrtmode	visszaállítja az eredeti képernyő üzemmódot.
setgraphbufsize	a grafikus buffer méretét határozza meg.
setgraphmode	kiválasztja és specifikálja a grafikus módot, törli a képernyőt és újratölti az összes adatot az alapértelmezés szerint.

Egy grafikus programot az ***initgraph*** függvény hívásával kell kezdeni, az ***initgraph*** betölti a grafikus meghajtót (.bgi állományt) és a rendszert a megfelelő grafikus módba helyezi.

Rajzolás

arc körívet rajzol.
circle kört rajzol.
drawpoly poligon körvonalát rajzolja.
ellipse ellipszis ívet rajzol.
getarccoords a körív vagy ellipszis ív utolsó hívásának koordináta értékeit adja vissza.
getaspectratio az aktuális grafikus mód vízszintes/függőleges képarányát adja meg.
line egyenest rajzol (x0,y0)-ból (x1,y1)-be.
linerel egyenes szakaszt rajzol egy pontba, relatív távolsággal a kurzor pozíciójától.
lineto egyenes szakaszt rajzol az aktuális pozíciótól az (x,y) pontba.
moveto mozgatja a kurzort az (x,y) pontba.
moverel mozgatja a kurzort egy relatív távolsággal.
rectangle téglalapot rajzol.
setaspectratio változtatja a figyelembe veendő képarányt.
setlinestyle beállítja az aktuális vonalvastagságot és rajzolási módot.

Kitöltés (fill)

bar rajzol és befest egy téglalapot.
bar3d rajzol és befest egy téglatestet.
fillellipse rajzol és befest egy ellipszist.
fillpoly rajzol és befest egy poligont.
floodfill befest egy zárt tartományt.
getfillpattern visszatér a beállított mintával.
getfillsettings visszatér az aktuális festő mintával és színével.
pieslice rajzol és befest egy körcikket.
sector rajzol és befest egy ellipszis cikket.
setfillpattern kiválasztja a felhasználó által definiált kitöltő mintát.
setfillstyle kiválasztja a kitöltő mintát és színt.

Képernyő- és ablakkezelés

cleardevice törli az aktív képernyőt (aktív lapot)
setactivepage kijelöli az aktív lapot grafikus outputra.

setvisualpage	kijelöli a látható grafikus lap számát.
clearviewport	törli az aktuális képernyő ablakot.
getviewsettings	visszatér a képernyő ablak információival.
setviewport	kijelöli az aktuális képernyő ablakot grafikus outputra.
getimage	a kijelölt tartományt memóriába menti ki.
imagesize	megadja a kijelölt téglalap alakú tartomány bájtjainak számát.
putimage	a korábban tárolt képernyő tartományt a képernyőre helyezi.
getpixel	megadja az (x,y) koordinátájú képpont színét.
putpixel	(x,y) koordinátájú pontban egy képpontot rajzol.

Szövegkiírás a képernyőre grafikus módban

gettextsettings	visszatér az aktuális karakterkészlet típusával, irányával, méretével és helyzetével.
outtext	az aktuális pozíciótól szöveget ír.
outtextxy	megadott pozíciótól szöveget ír.
settextjustify	a szöveg beállítási értékeket használja az ***outtext*** és ***outtextxy*** függvényeknél.
settextstyle	beállítja az aktuális karakterkészlet, a kiírás irányát és a karakterek méretét.
setusercharsize	beállítja a karakterek szélesség- és magasságfaktorát.
textheight	megadja a szöveg magasságát képpontokban.
textwidth	megadja a szöveg szélességét képpontokban.

Színbeállítás és lekérdezés

getbkcolor	megadja az aktuális háttérszínt.
getcolor	megadja az aktuális rajzolási színt.
getdefaultpalette	kitölti a paletta struktúrát.
getmaxcolor	az aktuális grafikus módban használható színek maximális értékét adja meg.
getpalette	az aktuális palettát és annak méretét adja meg.
getpalettesise	megadja a paletta színeinek számát.
setallpalette	változtatja a paletta színeit.
setbkcolor	beállítja az aktuális háttérszínt.
setcolor	beállítja az aktuális rajzolási színt.
setpalette	változtatja a paletta színeit az argumentum szerint.

Hibakezelés

grapherrormsg	a hibakódnak megfelelő szöveges hibaüzenetet adja.
graphresult	az utoljára végrehajtott grafikus művelet hibakódját adja vissza.

Ha hiba történik egy grafikus könyvtári függvény hívásakor, akkor a belső hibakód negatív értéket kap. Az utoljára végrehajtott grafikus művelet hibakódját a ***grapresult*** függvénnyel lekérdezhetjük. Ha a hibakód nulla, akkor sikeres volt a grafikus függvény végrehajtása. A ***graphresult*** függvénynek az alábbi kód konstansai léteznek:

Hiba-kód	Szimbólum	Magyarázat
0	*grOk*	Nincs hiba
-1	*grNoInitGraph*	Nincs .bgi f'ájl installálva
-2	*grNotDetected*	Nincs grafikus kártya
-3	*grFileNotFound*	A .bgi fájl hiányzik.
-4	*grInvalidDriver*	Érvénytelen grafikus meghajtó.
-5	*grNoLoadMem*	Kevés a memória a meghajtó számára.
-6	*grNoScanMem*	Kevés a memória a vizsgálathoz.
-7	*grNoFloodMem*	Kevés a memória a kitöltéshez.
-8	*grFontNotFound*	Nem létező karakterkészlet fájl.
-9	*grNoFontMem*	Kevés a memória betölteni a karakterkészletet
-10	*grInvalidMode*	A kiválasztott grafikus mód érvénytelen.
-11	*grError*	Grafikus hiba.
-12	*grIOerror*	Grafikus I/O hiba.
-13	*grInvalidFont*	Érvénytelen karakterkészlet fájl.
-14	*grInvalidFontNum*	Érvénytelen karakterkészlet azonosító.
-15	*grInvalidDeviceNum*	Érvénytelen egység szám.
-18	*grInvalidVersion*	Érvénytelen fájl verzió.

Állapotlekérdezés

getarccoords	az utoljára rajzolt körív vagy ellipszis ív koordinátáinak értékét adja meg.
getaspectratio	a grafikus képernyő oldalarány étékét adja meg.
getbkcolor	az aktuális háttér színét adja meg.

getcolor	az aktuális rajz színét adja meg.
getdrivername	az aktuális grafikus meghajtó nevét adja meg.
gerfillpattern	a felhasználó által definiált festő minta azonosítóját adja vissza.
getfillsettings	az aktuális festő mintát és színt adja meg.
getgraphmode	az aktuális grafikus módot adja meg.
getlinesettings	az aktuális vonal típusát, mintáját és vastagságát adja meg.
getmaxcolor	a maximálisan használható színek számát adja vissza.
getmaxmode	az aktuális meghajtó grafikus módjának maximális számát adja vissza.
getmaxx	a képernyő x irányú méretét adja vissza.
getmaxy	a képernyő y irányú méretét adja vissza.
getmodename	az aktuális mód nevét adja vissza.
getmoderange	az adott meghajtó módhatárait adja meg.
getpalette	a palettát és méretét adja meg.
gettextsettings	az aktuális karakterkészletet, irányát, méretét és helyzetét adja meg.
getviewsettngs	információt ad az aktuális rajzolási ablakról.
getx	az aktuális kurzorpozíció x koordináta értékét adja vissza.
gety	az aktuális kurzorpozíció y koordináta értékét adja vissza.

Színkezelés különböző meghajtók esetén

- EGA esetén a hardver 64 színt tud kezelni, azonban csak 16 színt használhatunk egyidejűleg.
- CGA grafikus módban (320x200) kisfelbontású grafikában 4 színt használhatunk, finom grafikában (640x200) pedig csak 2 színt.

A CGA kisfelbontású grafika színválasztéka

Kisfelbontású grafikában a 4 szín közül egy szín a háttér, amely a 16 szín közül bármely lehet. Négy paletta közül választhatunk ki további három színt a rajzoláshoz. A mód kiválasztásával aktiváljuk a palettát, amikor a CGA0, CGA1, CGAC2 vagy a CGAC3 között választunk.

Paletta szám	**1**	**2**	**3**
CGAC0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
CGAC1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WRITE
CGAC2	CGA_GREEN	CGA_RED	CGA_BROWN

Például, ha a 3-as palettát választottuk, azaz a grafikus üzemmódnak a CGAC3-at adtuk meg, akkor az alábbi színeket használhatjuk:

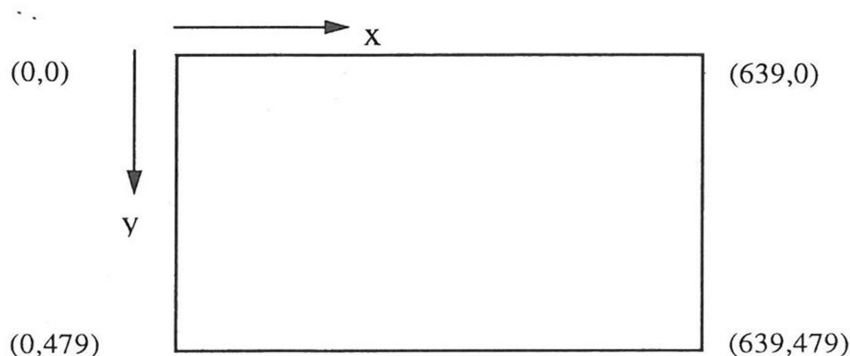
CGA_CYAN CGA_MAGENTA CGA_LIGHTGRAY

Az EGA és VGA grafika színválasztéka

Az EGA paletta 16 színt tartalmaz a lehetséges 64 színből. Az alapértelmezés szerinti EGA paletta megegyezik a 16 CGA színnel. A fekete szín kódja 0, a kék színé 1, stb. A graphics.h állományban definiált konstansokat felhasználva például az EGA_BLACK feketét, az EGA_WHITE fehéret jelent. A ***setbkcolor*** másképpen viselkedik EGA esetén, mint CGA-nál. A ***setbkcolor*** EGA esetén az aktuális szín értékét tárolja. A szín hivatkozás ugyanolyan VGA kártya esetén, mint EGA-nál, mert a VGA meghajtó EGA meghajtóhoz hasonlóan viselkedik, csak nagyobb a felbontása (kisebbek a képpont méretei).

Koordinátarendszer grafikus módban

Bármely grafikus módnál a képernyő bal felső sarka a (0,0) pont. Az x értéke (oszlopok) jobbra, az y értékek (sorok) lefelé növekednek. A sorokban és az oszlopokban a pontok színe függ a grafikus meghajtótól. A koordinátaértékek programból lekérdezhetők a ***getmaxx*** és a ***getmaxy*** függvényekkel.



4.2. ábra Koordináták a grafikus képernyőn

4.2. Grafikus programok és animáció

4.2.1. Halászás

Irjon programot, ahol a *hal* nevű objektum 10-30 közötti halat tart nyilván *x,y* koordinátáival, és a halak darabszámával (*n*). Egy-egy halat 5 pixel sugarú sárga kör reprezentál.
Tagfüggvényei:

init tagfüggvény a halak darabszámát sorsolja, valamint a halak *x,y* koordinátáit is adott határok közötti véletlenszámokkal tölti fel.
hatterszin tagfüggvény a hátteret türkíz színűre állítja be.
halrajz tagfüggvény kirajzolja a halakat reprezentáló sárga köröket.

A merítőhálót a *halo* objektum jelképezi, amely a *hal* objektumból származik.
Adattagjai:

xx,yy a merítőháló koordinátái. A háló egy 10 pixel sugarú piros kör. A merítőháló a nyíl iránygombokkal pozícionálható.

Metódusai:

init a hálót a 200,200 pontban helyezi,
up felfelé,
down lefelé,
left balra,
right jobbra mozgatja a hálót.
mozog kezeli a nyíl iránygombokat és figyeli a halhoz való közlítést, az ESC lenyomásával a program futása megszakad.

A feladat megoldását a HALASZP1.CPP tartalmazza.

```
/* halaszp1.cpp */
#include <iostream.h>
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>

#define balra   75
#define jobbra  77
#define fel     72
#define le      80
#define esc     27
```

```
class  hal
{
  public:
    int  n;
    int  x[30],y[30];
  public:
    hal();
    void hatterszin();
    void halrajz();
};
void hal::hatterszin()
{
  setbkcolor(CYAN);
}
hal::hal()
{
  int i;
  randomize;
  n = random(20)+10;
  for( i=1; i<n; i++)
  {
    x[i] = random(630)+10;
    y[i] = random(460)+10;
  }
}
void  hal::halrajz()
{
  int i;
  for( i=1; i<n; i++)
  {
    setcolor(YELLOW);
    if (x[i] > 0 && y[i] > 0)
      circle(x[i],y[i],5);
  }
}
class  halo :public hal
{
  public:
    int xx,yy;
  public:
    halo();
    void up();
    void down();
    void left();
    void right();
    void mozog();
 };
halo::halo():hal()
{
  xx = 200;
  yy = 200;
}
```

```
void  halo::up()
{
  cleardevice();
  if (yy>10)  yy = yy-1;
  setcolor(RED);
  circle(xx,yy,10);
}
void  halo::down()
{
  cleardevice();
  if (yy<469)  yy = yy+1;
    setcolor(RED);
    circle(xx,yy,10);
}
void halo::left()
{
  cleardevice();
  if (xx>10)  xx = xx-1;
  setcolor(RED);
  circle(xx,yy,10);
}
void halo::right()
{
  cleardevice();
  if (xx<629)  xx = xx+1;
  setcolor(RED);
  circle(xx,yy,10);
}
void halo::mozog()
{
  int key;
  int  i;
  hatterszin();
  setcolor(RED);
  circle(xx,yy,10);
  do
  {
    halrajz();
    key = getch();
    if (key==0)
            {
              key = getch();
              switch (key)
              {
               case 72: up(); break;
               case 80: down();break;
               case 75: left();break;
               case 77: right();break;
              }
            }
```

```
    for (i = 1; i<=n; i++)
    {
      if((abs(x[i]-xx)<15) && (abs(y[i]-yy)<15))
      {
       x[i] =0;
       y[i] =0;
      }
    }
  } while(key != esc);
}

  void grkezd()
  {
    int gd, gm;
    gd = VGA; gm = VGAHI;
    initgraph(&gd,&gm,"");
    if( graphresult()!= grOk)
    {
      cout << "Grafikus hiba!\n"; exit(1);
    }
  }
  void grvege()
  {
    closegraph();
  }
void main()
{
  halo  *h;
  grkezd();
  h = new halo();
  h->mozog();
  grvege();
  delete h;
}
```

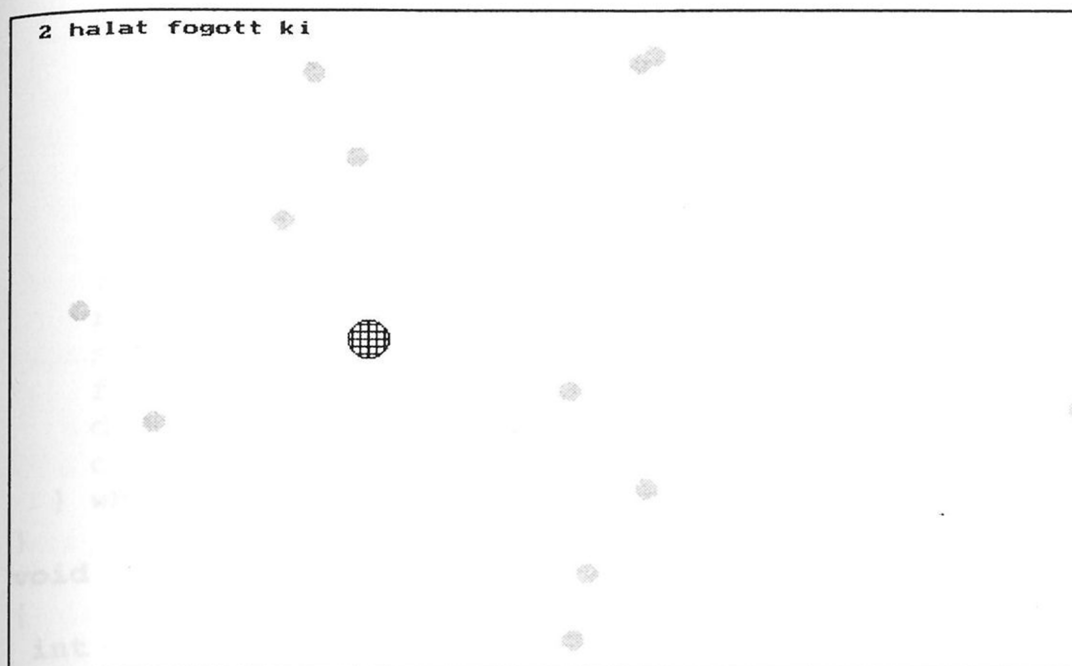
Módosítsuk a HALASZP1.CPP programot. A halak sárga telített kör, a háló pedig piros rácsos legyen. A hal kifogását hangjelzés is jelezze.
A feladat megoldását a HALASZP2.CPP tartalmazza.

```
/* halaszp2.cpp */
#include <iostream.h>
#include <stdio.h>
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <math.h>
#include <dos.h>

#define  balra   75
```

```
#define  jobbra  77
#define  fel     72
#define  le      80
#define  esc     27

class  hal
{
  public:
    int  n;
    int  x[30],y[30];
  public:
    hal();
    void hatterszin();
    void halrajz();
};
void hal::hatterszin()
{
  setbkcolor(CYAN);
}
hal::hal()
{
  int i;
  randomize;
  n = random(20)+10;
  for( i=1; i<n; i++)
  {
    x[i] = random(630)+10;
    y[i] = random(460)+10;
  }
}
void  hal::halrajz()
{
  int i;
  setcolor(YELLOW);
  setfillstyle(SOLID_FILL,YELLOW);
  for( i=1; i<n; i++)
  {
    if (x[i] > 0 && y[i] > 0)
       pieslice(x[i],y[i],0,360,5);
  }
}
class  halo :public hal
{
  public:
    int xx,yy;
    int db;
  public:
    halo();
    void up();
    void down();
    void left();
    void right();
```

```
    void mozog();
    int haldb();
 };
halo::halo():hal()
{
  xx = 200;
  yy = 200;
  db = 0;
}
void  halo::up()
{
  cleardevice();
  if (yy>10)  yy = yy-1;
  setcolor(RED);
  setfillstyle(HATCH_FILL,RED);
  pieslice(xx,yy,0,360,10);
}
void  halo::down()
{
  cleardevice();
  if (yy<469)  yy = yy+1;
    setcolor(RED);
    setfillstyle(HATCH_FILL,RED);
    pieslice(xx,yy,0,360,10);
}
void halo::left()
{
  cleardevice();
  if (xx>10)  xx = xx-1;
  setcolor(RED);
  setfillstyle(HATCH_FILL,RED);
  pieslice(xx,yy,0,360,10);
}
void halo::right()
{
  cleardevice();
  if (xx<629)  xx = xx+1;
  setcolor(RED);
  setfillstyle(HATCH_FILL,RED);
  pieslice(xx,yy,0,360,10);
}
void halo::mozog()
{
  int key;
  int  i;
  hatterszin();
  setcolor(RED);
  circle(xx,yy,10);
  do
  {
    halrajz();
    key = getch();
```

```
    if (key==0)
            {
              key = getch();
              switch (key)
              {
               case 72: up(); break;
               case 80: down();break;
               case 75: left();break;
               case 77: right();break;
              }
            }
    for (i = 1; i<=n; i++)
    {
      if(abs(x[i]-xx)<15 && abs(y[i]-yy)<15)
      {
       db++; sound(440);
       delay(300);
       nosound();
       x[i]=0;
       y[i]=0;
      }
    }
  } while(key != esc);
}
int halo::haldb()
{
    return db;
  }
  void grkezd()
  {
    int gd, gm;
    gd = VGA; gm = VGAHI;
    initgraph(&gd,&gm,"");
    if( graphresult()!= grOk)
    {
      cout << "Grafikus hiba!\n"; exit(1);
    }
  }
  void grvege()
  {
    closegraph();
  }
void main()
{
  halo  *h;
  grkezd();
  h = new halo();
  h->mozog();
  grvege();
  cout << "A kifogott halak száma: " << h->haldb();
  delete h;
}
```



4.2.2. Fénysorompó animációja

Definiáljon fénysorompó működését szimuláló objektumot, amely két piros lámpa villogását, majd fehér lámpára váltást oldja meg. A lámpák működésénél
- a piros villogások számát,
- a piros villogások közötti késleltetést, és
- a fehér lámpa késleltetését

a billentyűzetről olvassa be.

A feladat megoldását az UTATJAR2.PAS tartalmazza.

```
/* utatjar2.cpp */
// fénysorompó kirajzolása
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include <dos.h>
   class sorompo
   {
    public:
     int  villogas_db; // piros villogás száma
     int  piros_kesleltet; // piros villogás késleltetése
     int  feher_kesleltet; // fehér lámpa késleltetése
   public:
     void init(int v0, int pk0, int fk0);
```

```
        void rajzol();
        void bal();
        void jobb();
        void mukodes();
 };
void sorompo::init(int v0, int pk0, int fk0)
{
   villogas_db     = v0;
   piros_kesleltet = pk0;
   feher_kesleltet = fk0;
   rajzol();
}
void sorompo::rajzol()
{
  struct arccoordstype v1,v2,v3;
  setcolor(RED);
  circle (150,100,15);      // vörös optika
  circle (200,100,15);
  setcolor(WHITE);
  circle (175,140,15);      // fehér optika
  arc (150,100,90,210,30); // bal lekerekítés
  getarccoords (&v1);
  arc (200,100,330,90,30); // jobb lekerekítés
  getarccoords (&v2);
  arc (175,140,210,320,30);// alsó rész
  getarccoords (&v3);

  line(v1.xstart,v1.ystart,v2.xend,v2.yend); // felső
  line(v3.xstart,v3.ystart,v1.xend,v1.yend); // bal
  line(v2.xstart,v2.ystart,v3.xend,v3.yend); // jobb
  rectangle (170,170,180,400);               // árbóc
}

void sorompo::bal()
{
  setfillstyle(INTERLEAVE_FILL,RED);
  floodfill(150,100,RED);
}

void sorompo::jobb()
{
  setfillstyle(INTERLEAVE_FILL,RED);
  floodfill(200,100,RED);
}
void sorompo::mukodes()
{
  int i;
  do
  {
    for( i=1; i<= villogas_db; i++)
    {
      rajzol();
```

```
        bal();
        delay (piros_kesleltet);
        cleardevice();
        rajzol();
        jobb();
        delay (piros_kesleltet);
        cleardevice();
      }
      rajzol();
      setfillstyle(9,15);
      floodfill (175,140,15);
      delay(feher_kesleltet);
      cleardevice();
    } while (!kbhit());
}
void main()
{
 int      gd = VGA, gm = VGAHI, i;
 sorompo s;
 int      psz,pkesl,fkesl,error;
  cout <<  "A piros villogások száma: ";
  cin >>  psz;
  cout << "A piros villogások közötti késleltetés: ";
  cin >> pkesl;
  cout << "A fehér lámpa időtartama: ";
  cin >> fkesl;
  initgraph(&gd,&gm,"");
  error = graphresult();            // Megadja az átváltás eredményét
  if (error != grOk)                // Alapértelmezés szerint  grok=0
  {
    cout << "Grafikus hiba ! : "; grapherrormsg(error);
       // Kiírja a megfelelő hibaüzenetet
  }
  s.init(psz,pkesl,fkesl);
  s.mukodes();
  getch();
  closegraph();
}
```

Használjon dinamikus helyfoglalású adattagokat és objektumpéldányt!
A feladat megoldását az UTATJARM.PAS tartalmazza.

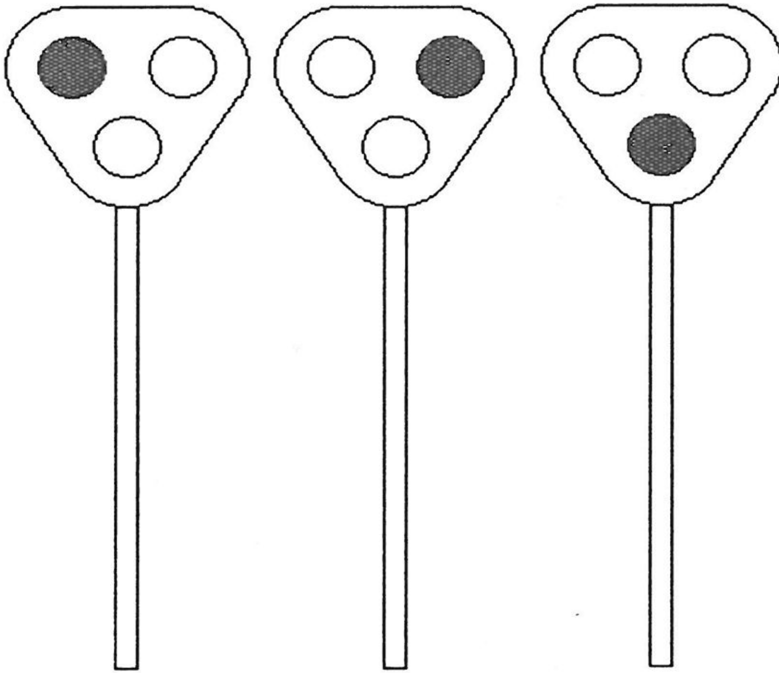
```
/* utatjarm.cpp */
// fénysorompó kirajzolása
#include <iostream.h>
#include <graphics.h>
#include <conio.h>
#include <dos.h>
```

```
    class  sorompo
    {
      public:
        int  *villogas_db; // piros villogás száma
        int  *piros_kesleltet; // piros villogás késleltetése
        int  *feher_kesleltet; // fehér lámpa késleltetése
     public:
        sorompo();
        ~sorompo();
        void init(int v0, int pk0, int fk0);
        void rajzol();
        void bal();
        void jobb();
        void mukodes();
  };
sorompo::sorompo()
{
   villogas_db = new int;
   piros_kesleltet = new int;
   feher_kesleltet = new int;
}
sorompo::~sorompo()
{
  delete villogas_db;
  delete piros_kesleltet;
  delete feher_kesleltet;
}
void sorompo::init(int v0, int pk0, int fk0)
{
   *villogas_db     = v0;
   *piros_kesleltet = pk0;
   *feher_kesleltet = fk0;
   rajzol();
}
void sorompo::rajzol()
{
  struct arccoordstype v1,v2,v3;
  setcolor(RED);
  circle (150,100,15);      // vörös optika
  circle (200,100,15);
  setcolor(WHITE);
  circle (175,140,15);      // fehér optika
  arc (150,100,90,210,30); // bal lekerekítés
  getarccoords (&v1);
  arc (200,100,330,90,30); // jobb lekerekítés
  getarccoords (&v2);
  arc (175,140,210,320,30);// alsó rész
  getarccoords (&v3);

  line(v1.xstart,v1.ystart,v2.xend,v2.yend); // felső
  line(v3.xstart,v3.ystart,v1.xend,v1.yend); // bal
  line(v2.xstart,v2.ystart,v3.xend,v3.yend); // jobb
```

```
  rectangle (170,170,180,400);                    // árbóc
}

void sorompo::bal()
{
  setfillstyle(INTERLEAVE_FILL,RED);
  floodfill(150,100,RED);
}

void sorompo::jobb()
{
  setfillstyle(INTERLEAVE_FILL,RED);
  floodfill(200,100,RED);
}
void sorompo::mukodes()
{
  int i;
  do
  {
    for( i=1; i<= *villogas_db; i++)
    {
      rajzol();
      bal();
      delay (*piros_kesleltet);
      cleardevice();
      rajzol();
      jobb();
      delay (*piros_kesleltet);
      cleardevice();
    }
    rajzol();
    setfillstyle(9,15);
    floodfill (175,140,15);
    delay(*feher_kesleltet);
    cleardevice();
  } while (!kbhit());
}
void main()
{
 int      gd = VGA, gm = VGAHI, i;
 sorompo *s;
 int      psz,pkesl,fkesl,error;
  cout <<  "A piros villogások száma: ";
  cin >>  psz;
  cout << "A piros villogások közötti késleltetés: ";
  cin >> pkesl;
  cout << "A fehér lámpa időtartama: ";
  cin >> fkesl;
  initgraph(&gd,&gm,"");
  error = graphresult();          // Megadja az átváltás eredményét
  if (error != grOk)              // Alapértelmezés szerint  grok=0
  {
```

```
    cout << "Grafikus hiba ! : "; grapherrormsg(error);
       // Kiírja a megfelelô hibaüzenetet
  }
  s = new sorompo();
  s->init(psz,pkesl,fkesl);
  s->mukodes();
  getch();
  closegraph();
  delete s;
}
```

A program futásának eredménye:



F1. Új nyelvi elemek a Borland C++ 5.02 implementációban

A következőkben összefoglaljuk azokat a nyelvi elemeket, amelyek újdonságot jelentenek a könyvünk alapjául szolgáló a Borland C++ 5.02 verzióban a korábbi Borland C++ verziókhoz képest. Összehasonlításunkban a DOS alatt futó (de Windows 3.x alkalmazások fejlesztésére is alkalmas) Borland C++ 3.1 verziót és a Windows'95-re kifejlesztett Borland C++ 5.02-es verziót tekintettük mérvadónak.

F1.1. Logikai típus

Logikai típusú változók definiálására használhatjuk a **bool** előredefiniált típust. Ilyen, **bool** típusú eredményeket szolgáltatnak a logikai operátorok illetve a relációs műveletek. A **bool** kulcsszó tehát olyan típust reprezentál, amelyeknek az értéke az előredefiniált **false** vagy **true** lehet. A **false** konstans numerikus értéke 0, míg **true** értéke 1. Ezek a logikai típusú konstansok csak jobbérték kifejezésként használhatóak. Egy **bool** típusú jobbérték kifejezés egész típusú jobb értékké konvertálható. A numerikus konvertálás után tehát **false**-ból 0, míg **true**-ból 1 lesz. Tetszőleges aritmetikai, felsorolt típusú vagy mutató típusú jobb érték kifejezés bool típusú jobbértékké konvertálható. A 0 vagy NULL pointert érték **false**-sá konvertálódik, minden más értékből **true** lesz.

F1.2. Új konverziós operátorok

Ebben a szakaszban a típusmódosítás újfajta alternatív módszereit tárgyaljuk. Ezek a módszerek kiegészítik a C nyelv korábbi típusmódosító operátorait.

const_cast

Alakja:

```
const_cast < T > (arg)
```

A **const_cast** operátor hozzáadja vagy leveszi a **const** illetve a **volatile** módosítók hatását a megadott típus kapcsán. Ez a típusmódosítás fordítási időben történik. A **const_cast** használatakor *T* és *arg* azonos típusok kell hogy

legyenek, eltekintve a **const** vagy **volatile** módosító használatától. Tetszőleges számú **const** vagy **volatile** módosítás végezhető egyetlen egy **cons_cast** utasítással. Egy konstansra mutató kifejezés nem konstansra mutató kifejezéssé alakítható. Sikeres konverzió esetén is a pointer ugyanarra az objektumra fog mutatni. Ugyanez igaz a referencia típusokra is. Hasonlóan működik a **const_cast** a **volatile** módosító esetében.

dynamic_cast

Alakja:

```
dynamic_cast < T > (ptr)
```

T-nek egy osztályra mutató pointernek vagy referenciának, illetőleg **void*** típusúnak kell lennie, míg a *ptr* argumentum egy pointer vagy referencia típusú kifejezés lehet. Ha *T* **void*** típusú, akkor *ptr*-nek szintén pointer típusúnak kell lenni. Ebben az esetben a keletkező pointer a szóban forgó osztály származási fájának tetején lévő osztály minden elemét elérheti. Egy ilyen osztály nem lehet további származtatott osztályok alap osztálya.

Származtatott osztályból alaposztályba vagy egyik osztályból egy másik osztályba a következőképpen konvertálhatunk: ha *T* egy pointer és *ptr* egy osztályhierarchiában lévő (nem alaposztály) osztályra mutató pointer, akkor az eredmény az egyedi alosztályra mutató pointer lesz. Hivatkozási típusokat hasonló módon kezel az operátor. Sikeres konverzió esetén a *ptr* a kívánt típusú lesz. Sikertelen konverzió esetén *ptr* értéke 0 lesz és *Bad_cast* kivétel jön létre. Fontos, hogy a **dynamic_cast** végrehajtásához ún. futási-idejű típusazonosításra (*run time type identification* - RTTI) van szükség.

reinterpret_cast

Alakja:

```
reinterpret_cast < T > (arg)
```

Ebben az utasításban a *T* mutató, referencia, numerikus típus, függvénymutató, vagy osztálytagra mutató pointer lehet. Ezen operátor segítségével egy mutatót egész típusúvá, illetve egy egész típusú kifejezést mutatóvá konvertálhatunk Az oda-vissza konvertálás ugyanazt az eredményt adja. A **reinterpret_cast** művelettel egy, még definiálatlan osztály felhasználható egy pointer vagy hivatkozás konverzió során.

Egy függvénypointer tetszőleges objektum mutatóvá konvertálható, feltéve, hogy a másik mutatótípus elegendő bitszélességű ahhoz, hogy egy függvénycímet tároljon.

static_cast

Alakja:

```
static_cast < T > (arg)
```

A *T* mutató-, referencia-, numerikus típus vagy **enum** típusú lehet. Az *arg* típusának meg kell egyezni *T* típusával. Mind *T*, mint *arg* típusának fordítási időben ismertnek kell lenni.

F1.3. Futásidejű típusazonosítás

A **typeid** operátornak kétféle alakja van:

```
typeid (kifejezés)
```

vagy

```
typeid (típusazonosító)
```

A **typeid** operátor segítségével adattípusok illetve kifejezések típusazonosítása a program futása közben is lekérdezhető legyen. A **typeid** operátor használata esetén egy **typeinfo** típusú objektumra vonatkozó referenciát kapunk. Az így elérhető objektum a **typeid** operandusának típusát jellemzi. A **typeinfo** típus a typeinfo.h fejlécfájlban van definiálva. A **typeid** operátor használatát az alábbi egyszerű példa szemlélteti:

```
if (typeid(x) == typeid (y)) cout << "x és y azonos típusúak ";
```

F1.4. A typename kulcsszó

A **typename** kulcsszó a típussablonok használatához kötődik. Kétféle felhasználási módja van: Definiálatlan osztályok esetében a **class** kulcsszó helyett használható **typedef** utasításokban, illetve sablon-deklarációkban a **class** kulcsszó helyett állhat.

F1.5. Kivételek kezelése

Ha egy program futása során abnormális szituáció adódik, akkor a program futása befejeződhet. Az ilyen kivételes szituációk kezelését röviden kivételkezelésnek nevezzük. A C++-ban megvalósított kivételkezelésnek három eleme van:

- kivétel kezelés alatt álló programrészlet definiálása (**try** blokk),
- kivételek továbbítása (**throw**),
- kivételek elkapása és lekezelése (**catch**).

A kivételkezelést tehát 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** *kifejezés* utasítással adhatjuk át a vezérlést a kifejezés típusának megfelelő kezelőnek (*handler*), amelyet a **catch** kulcsszót követően adunk meg.

```
try{
                        // a kivétel figyelés alatt álló utasítások
}
catch (kivétel_deklaráció_1)
{                       // a kivétel_1 kezelője
     utasítások;
}
catch (kivétel_deklaráció_2)
{                       // a kivétel_2 kezelője
     utasítások;
}
```

Ha egy kivétel függvényen belül keletkezik, akkor a függvény fejlécében kijelölhetjük, hogy mely kivételek továbbítódjanak a függvényen kívüli kezelőhöz:

```
int fv1();    // minden kivétel továbbítódik

int fv2() throw(char *);
              // csak char* típusú kivételek továbbítódnak

int fv3() throw();
              // egyetlen kivétel sem továbbítódik
```

A beérkező kivételek a típusok alapján a megfelelő kezelőhöz irányítódnak:

```
catch (char *s)     // a char* kivétel kezelője
{
        // a kezelő törzse
}

catch (...)  // minden kivétel kezelője
{
        // a kezelő törzse
}
```

Amikor egy kivételt továbbítunk a **throw** utasítással, akkor a megadott kifejezés értéke átmásolódik a **catch** utasítással megadott 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. Ha egy olyan kivételt továbbítunk, amelyik nincs benne az adott függvény által továbbítandó kivételek listájába, akkor az ***unexpected*** rendszerhívás állítja le a programunk futását. Mindkét kilépési folyamatba beavatkozhatunk saját kezelők definiálásával, melyek regisztrációját az except.h állományban deklarált ***set_terminate*** illetve ***set_unexpected*** függvényekkel végezhetjük el.

példa a kivételkezelésre

F1.6. Nevek érvényességének korlátozása

A legtöbb komoly alkalmazói program számos forrásállományból áll. Ezen állományokat több programozó fejleszti illetve tartja karban. Ezen különálló állományok úgy vannak szervezve, hogy belőlük összeállítható legyen a végső alkalmazói program. Hagyományosan a fájlokat úgy szervezik, hogy minden olyan név (azonosító), amelyik nincs egy adott ún. név-térbe vagy névterületbe (ilyen például egy függvény- vagy osztálydefiníció törzse ill. egy fordítási egység) azonos név-teret kell, hogy megosszon egymással. Éppen ezért a különböző programmodulok összeszerkesztése során felfedezett azonos neveket egymástól valamilyen módon meg kell különböztetni. A globális nevek ilyen ütközésének feloldására a C++ nyelv egy névterület-kezelési mechanizmust kínál a **namespace** kulcsszó segítségével. Ez a mechanizmus lehetővé teszi, hogy a nevek szempontjából egy alkalmazói programot több különböző alrendszerre bontsunk. Minden ilyen alrendszer a saját hatáskörében tetszőleges neveket használhat. Ilyen módon minden programfejlesztő

szabadon választhatja meg az általa használt azonosítókat. Így a programozóknak nem kell törődniük névütközési problémákkal. Ezen mechanizmus első lépése egy egyedi azonosítóval ellátott név-tér definiálása a **namespace** kulcsszó segítségével. A második lépés az, hogy a **using** kulcsszóval megadjuk melyik névterületen definiált azonosítókat kívánjuk használni. Egy **namespace** deklarációban egy olyan azonosítót adhatunk csak meg, amelyet korábban nem használtunk globális azonosítóként. Például:

```
namespace ALFA {
    // ALFA a név-tér azonosítója
    // tetszőleges deklarációk
}
```

Egy **namespace** azonosító minden olyan fordítási egységben ismernek kell lenni, amelyben elemeit fel kívánjuk használni.

Hosszú név-tér azonosítók helyett alternatív (alias) azonosítók is használhatók:

```
namespace EZEGYHOSSZUAZONOSITO {

        // tetszőleges deklarációk
}
// alternatív azonosító
namespace ROVIDEBB = EZEGYHOSSZUAZONOSITO;
```

A **using** direktíva segítségével megadjuk azt a névterületet, amelynek az azonosítóit használni szeretnénk:

```
using namespace ELSO;
using namespace MASODIK;
// az ELSO és MASODIK név-térben definiált azonosítók
// használhatók ettől a ponttól kezdve
```

A helyi érvényességű nevek továbbra is elfedik az azonos külső neveket.

F1.7. Nagy bitszélességű karaktertípus

A C++ programokban a **wchar_t** egy olyan alaptípus, amelyik (a helyileg támogatott) legnagyobb méretű kiterjesztett karakterkészlet elemeinek ábrázolására alkalmas. A **wchar_t** típus mérete, előjel értelmezése és bájt sorrendje ugyanaz, mint az **int** típusé. Megjegyzés: C programok az stddef.h fejléc fájl alkalmazásával **wchar_t** konstansokat használhatnak. A C++-ban a **wchar_t** kulcsszóként szerepel.

F1.8. A mutable kulcsszó

A **mutable** módosító arra használható, hogy egy változó módosítható legyen annak ellenére, hogy egy konstans kifejezés része. Csak osztályok adattagjai lehetnek **mutable** módosítóval deklarálni. A **mutable** kulcsszó nem használható **static** és **const** nevekre. A **mutable** kulcsszó célja az, hogy specifikáljuk, mely adattagokat módosíthat egy **const** tagfüggvény. Alapesetben egy **const** tagfüggvény nem módosíthat adattagokat.

F1.9. Konstruktorok szigorú paraméterezése

Általában az egyparaméteres konstruktorral rendelkező osztályok példányainak olyan kifejezést adhatunk értékül, amely típusával illeszkedik a konstruktor paraméterének típusához. Ekkor a kifejezés értéke automatikusan egy olyan típusú objektumpéldánnyá konvertálódik, amilyen típusú objektumnak értékül kívánjuk adni. Ezt a fajta implicit konverziót megakadályozhatjuk, ha a konstruktort az **explicit** kulcsszóval deklaráljuk. Ekkor a kérdéses osztály minden példányának csak olyan érték adható, amelynek típusa megegyezik az osztálytípusával. Tekintsük a következő példát:

```
class valami
{
   ...
   public:
    valami (int a);                       // első konstruktor
    valami (const char* b, int a = 0); // második konstruktor
    ...
};

valami x = 1;
valami y = "alma";
```

Az *x* objektum az első konstruktor szerinti implicit konverzióval jön létre egy egész típusú kifejezésből, míg az *y* objektum a második konstruktor szerinti konverzióval jön létre egy sztringkonstansból. Ha azonban a

```
class valami
{
   ...
   public:
    explicit valami (int a);     // első konstruktor
    explicit valami (const char* b, int a = 0);
                                  // második konstruktor
    ...
};
```

deklarációt alkalmazzuk, akkor csak a következőképpen adhatunk értéke az x és y objektumpéldányoknak:

```
valami x = valami(1);
valami y = valami("alma",2);
```

F2. Objektum-orientált megközelítésű függvények

A függvény overloading és az operátor overloading tette lehetővé, hogy a Borland C++-ban – a FORTRAN-hoz hasonlóan – használhassunk komplex számokat, és hogy BCD ábrázolású számokkal is dolgozhassunk. A Borland C++ például a komplex számok ábrázolására definiálja a complex osztályt, és kiterjeszti az összes aritmetikai operátor, valamint a standard matematikai függvények jelentését a complex típusú adatokra. Hasonlóképpen, a BCD számok ábrázolására definiálták a *bcd* osztályt, és az aritmetikai operátorok értelmezését erre a típusra is kiterjesztették. A következőkben e típusokkal ismerkedünk meg részletesen.

F2.1. Komplex aritmetika

Egy komplex szám

$$x + i \cdot y$$

alakú, ahol x és y valós számok és $i^2 = -1$. A Borland C++-ban az alábbi struktúrát definiálták a math.h állományban:

```
struct complex
{
    double x,y;
}
```

mely mindig hozzáférhető – akár C-ben, akár C++-ban dolgozunk. Ez azonban még nem elegendő ahhoz, hogy komplex műveleteket végezhessünk. Ha csak C-ben dolgozunk, akkor az előbbi struktúrával nem sokat érünk, a ***cabs*** függvény kivételével más műveletek nem igen végezhetünk, mert hiányoznak a megfelelő definíciók. A problémát a C++ által nyújtott *operátor-overloading* oldja meg. Ha a complex.h állományt építjük be programunkba, akkor a complex típus **class** deklarációja – és ezzel együtt a komplex számokra a Borland C++-ban értelmezett minden függvény – a rendelkezésünkre áll, azaz:

- használhatjuk az összes aritmetikai műveleteket (+,-,*,/, +=, -=, *=, /=, = ==, !=)
- a stream (folyam) operátorokat (>>, <<)
- a gyakran használt matematikai függvényeket:

abs

```
friend double abs(complex& val);
```

Visszatér a komplex szám abszolút értékével.

acos

```
friend complex acos(complex& z);
```

Kiszámítja az arkusz koszinuszát.

arg

```
double arg(complex x);
```

A komplex síkon lévő számnak a szögét radiánban adja meg.

asin

```
friend complex asin(complex z);
```

Kiszámítja az arkusz szinuszát.

atan

```
complex atan(complex& z);
```

Kiszámítja az arkusz tangenst.

conj

```
friend complex conj(complex& z);
```

Egy komplex szám konjugált komplex számát adja vissza.

cos

```
friend complex cos(complex& z);
```

Kiszámítja az érték koszinuszát,

cosh

```
friend complex cosh(complex& z);
```

Kiszámítja az érték hiberbolikusz koszinuszát

exp

```
friend complex exp(complex& y);
```

Kiszámítja az érték exponenciálisát.

imag

```
double imag(complex x);
```

Visszatér a komplex szám imaginárius értékével.

log

```
friend complex log(complex& z);
```

Visszatér az *z* természetes logaritmusával.

log10

```
friend complex log10(complex& z);
```

norm

```
double norm(complex x);
```

Visszatér az abszolút érték négyzetével.

polar

```
complex polar(double mag, double angle = 0);
```

Visszatér a komplex szám abszolút értékével és szögével.

pow

```
friend complex pow(complex& base, double expon);
friend complex pow(double expon, complex& base,);
friend complex pow(complex& base, double& expon);
```

Kiszámítja a *base* komplex szám *expon* hatványra emelt értékét.

real

```
long double real (bcd number);
double real(complex x);
```

Konvertálja a BCD vagy komplex számot **long double**-ra vagy visszatér a komplex szám valós részével.

sin

```
friend complex sin(complex& z);
```

Kiszámítja a szinuszát.

sinh

```
friend complex sinh(complex& z);
```

Kiszámítja a szinusz hiperbólikuszát.

sqrt

```
friend complex sqrt(complex& z);
```

Kiszámítja a pozitív négyzetgyökét.

tan

```
friend complex tan(complex& z);
```

Kiszámítja a tangensét.

tanh

```
friend complex tanh(complex& z);
```

Kiszámítja a tangens hiperbólikuszát.

F2.2. BCD aritmetika

A Borland C++ hasonlóan a többi fordító programokhoz az aritmetikai műveleteket kettes számrendszerben végzi el. Ennek annyi hátránya van, hogy a legtöbb 10-es számrendszerben pontosan ábrázolható szám (mint például 0.01) a 2-es számrendszerben csak közelítő pontossággal ábrázolható.

A bináris számábrázolás kedvező a legtöbb számításnál, de sok esetben a kerekítési hibák kumulálódnak az elengedhetetlen konverzióknál. Elképzelhető, hogy a kerekítési hibák kis számoknál 0 eredményt szolgáltatnak. A problémát elodázhatjuk, ha **double** vagy long **double** típusú változókkal számolunk, de előbb-utóbb előjön a 10-es számrendszerbeli valós számok véges pontosságú bináris reprezentációjának a kérdése.

Ezt a problémát áthidalhatjuk, ha az ún. binárisan kódolt decimális számábrázolást (*Binary Coded Decimal* = BCD) használhatjuk. Nos, a Borland C++ -ban erre lehetőségünk van a bcd.h állományban deklarált bcd típus által. Így például a 0.01 is pontosan ábrázolható, ha *bcd* típusú változatban tároljuk, és így végzünk műveleteket vele. A *bcd* típus használata során is figyelembe kell vennünk néhány szempontot. Például

- A *bcd* ábrázolás sem tud minden kerekítési hibát megszüntetni. Például az 1.0/3.0 kifejezés értéke végtelen tizedes tört, így a *bcd*-ben is csak kerekítve ábrázolható.
- A gyakran használt matematikai függvényeknél ilyen például az ***sqrt*** és a ***log*** – a *bcd* argumentum felíródik.
- A *bcd* számok 17 decimális jegyre pontosak, az ábrázolási pontosságuk határa: 10^{-125} és 10^{125} .

A *bcd* típus különbözik a **float, double** vagy a **long double** típusoktól. Ha egy kifejezés egy tagja *bcd* ábrázolású, akkor automatikusan az aritmetikai műveletek *bcd* változata kerül végrehajtásra. A *bcd* típusnak (amelyik termé-

szetesen egy **class**), több konstruktora létezik. Ezek segítségével állíthatunk elő *bcd* számokat különböző elemi típusú kifejezésekkel.

Konstruktorai:

Alapértelmezés:	`bcd();`
int típusú	`bdc(int x);`
Előjel nélküli **int**	`bcd(unsigned int x);`
Long típusú	`bcd(long x);`
Előjel nélküli **long**	`bcd(unsigned long x);`
Double típusú	`bcd(double x, int decimals = Max);`
Hosszú **double**	`bcd(long double x, int decimlas = Max);`
	A *decimalis* paraméter a decimális jegyek száma.

A *bcd*-bináris konverzió számára megadható a figyelembe veendő decimális jegyek száma.

```
1000.00/7             = 142.8514...
bcd(1000.00/7,2)      = 142.860
bcd(1000.00/7,1)      = 142.900
bcd(1000.00/7,0)      = 143.000
bcd(1000.00/7,-1)     = 140.000
bcd(1000.00/7,-2)     = 100.000
```

A kerekítés a legközelebbi egész számra történik, ha a szám 5-re végződik, a kerekített jegy páros lesz.

```
bcd(12.335, 2)        = 12.34
bcd(12.345, 2)        = 12.34
bcd(12.355, 2)        = 12.36
```

Friend függvények

A *bcd* számokat bármely ANSI C alapvető matematikai függvényben használhatjuk. A következő ANSI C matematikai függvények lesznek felülbírálva, hogy a *bcd* típussal működjenek:

```
friendbcdabs(bcd&);
friendbcdacos(bcd&);
friendbcdasin(bcd&);
friendbcdatan(bcd&);
friendbcdcos(bcd&);
friendbcdcosh(bcd&);
friendbcdexp(bcd&);
```

```
friendbcdlog(bcd&);
friendbcdlog10(bcd&);
friendbcdpow(bcd&base,bcd&expon);
friendbcdsin(bcd&);
friendbcdsinh(bcd&);
friendbcdsqrt(bcd&);
friendbcdtan(bcd&);
friendbcdtanh(bcd&);
```

Operátorok

A *bcd* osztály felülbírálja az operátorokat :

+,-,*,/,+=,-=,*=,/=,=,==,!=

A *bcd* számokra is használható a << és a >> a bemeneti és kimeneti adatfolyam.

F3. Függvények szöveges üzemmódban

clreol **conio.h**

Törli az összes karaktert a kurzortól sor végéig a képernyő ablakban, anélkül, hogy a kurzort elmozgatná.

```
void clreol(void);
```

Megjegyzés: Törlés után a sor a kurzor helyétől az aktuális képernyő ablak széléig háttérszínű lesz.

clrscr **conio.h**

Törli a képernyő ablakot:

```
void clrscr(void).
```

Megjegyzés: Ha a háttér nem volt fekete, akkor a törlés után a háttér felveszi az előzőekben definiált háttér színét. A kurzor áthelyeződik a képernyő bal felső sarkába (1,1).

delline **conio.h**

Törli a kurzort tartalmazó sort.

```
void delline(void);
```

Megjegyzés: A ***delline*** törli a kurzort tartalmazó sort és az alatta lévő sorok egy ponttal feljebb lépnek a képernyőn. A ***delline*** az aktuális képernyő ablakon belül működik.

gettext **conio.h**

Szöveget másol a képernyőről memóriaterületre.

```
int gettext(int left, int top, int right, int bottom,
            void destin);
```

Paraméterek:
left,top a téglalap bal felső sarka
right,bottom a téglalap jobb alsó sarka
destin memória területre mutató pointer

Megjegyzés: A ***gettext*** a téglalap alakú képernyő tartalmát másolja a megadott memóriaterületre. Minden koordinátát abszolút koordinátaértékkel kell megadni. Minden karakter tárolására 2 bájt szükséges, ezért h sor és v oszlop tárolására

$$\text{bytes} = (\text{h} \cdot \text{sor}) \cdot (\text{w} \cdot \text{oszlop}) \cdot 2$$

darab bájt szükséges. Sikeres végrehajtás esetén a visszatérési érték 1, különben 0.

gettextinfo	**conio.h**

A text üzemmódról nyújt információt.

```
void gettextinfo(struct text_info *r);
```

Paraméter:
*r struktúrára mutató pointer - ez adja meg a text video információkat.
Megjegyzés: A *text_info* struktúra a conio.h-ban van definiálva.

```
struct text_info
{
    unsigned char winleft;       /* az ablak bal koordinátája       */
    unsigned char wintop;        /* az ablak felso koordinátája     */
    unsigned char winright;      /* az ablak jobb koordinátája      */
    unsigned char winbottom;     /* az ablak alsó koordinátája      */
    unsigned char attribute;     /* szöveg attributum               */
    unsigned char normattr;      /* normal attributum               */
    unsigned char currmode;      /* BW40, BC80, C40, vagy C80       */
    unsigned char screenheight;  /* text képenyő magassága          */
    unsigned char screenwidth;   /* text képernyő szélessége        */
    unsigned char curx;          /* aktuális ablak x koordinátája   */
    unsigned char cury;          /* aktuális ablak y koordinátája   */
};
```

gotoxy **conio.h**

Pozícionálja a kurzort

```
void gotoxy(int x, int y);
```

Paraméter:

x,y a kurzor új pozíciója.

Megjegyzés: Az aktív ablak adott pozíciójába mozgatja a kurzort, az *x*-edik oszlopba és az *y*-adik sorba. Az ablak bal felső sarokpontja (1,1) . Érvénytelen koordináták esetén a gotoxy hívás nem kerül végrehajtásra. Például a kurzort a 10. oszlopba és a 20. sorba pozícionálja a gotoxy(10,20); függvényhívás.

highvideo **conio.h**

Intenzív (magas fényű) karakterszínkiválasztását végzi.

```
void highvideo(void);
```

Megjegyzés: A ***highvideo*** a nagy intenzitást az aktuálisan kiválasztott háttérszín legmagasabb helyiértékű bitjének beállításával valósítja meg.

insline **conio.h**

Beszúr egy üres sort a kurzor pozíciónál.

```
void insline(void);
```

Megjegyzés: A sorbeszúrás következtében a utolsó sor kilép a képernyőből.

lowvideo **conio.h**

Normál (alacsony fényű) karakterszín kiválasztását végzi.

```
void lowvideo(void);
```

Megjegyzés: A ***lowvideo*** törli az aktuálisan kiválasztott háttérszín legmagasabb helyiértékű bitjét, amely a magasabb intenzitást jelölte.

movetext **conio.h**

Szöveget másol át egyik téglalapból a másikba.

```
int movetext(int left, int top, int right,
             int bottom, int destleft, int desttop);
```

Paraméterek:

left, top	a forrás téglalap bal felső sarka
right, bottom	jobb alsó sarka
destleft, desttop	a célterület bal felső sarka

Megjegyzés: Az egyik téglalapból a másik téglalapba másolja a képernyő tartalmát. Például legyen az egyik téglalap bal felső sarkának koordinátája (5,15), a jobb alsó sarkának koordinátája (20,25), másoljuk át egy új területre, melynek a bal felső koordinátái (30,5)! Ext a

```
movettext(5,15,20,25,30,5);
```

függvényhívással tehetjük meg.

normvideo **conio.h**

Normál intenzitású karakterszín kiválasztását végzi.

```
void normvideo(void);
```

Megjegyzés: Visszatölti azt a szöveg attribútumot, amelyet az indításnál használt a program.

puttext **conio.h**

Memóriából képernyőre másol.

```
int puttext(int left, int top, int right,
            int bottom, void *source);
```

Paraméterek:

left,top	a téglalap bal felső sarka
right,bottom	a téglalap jobb alsó sarka
source	memória területre mutató pointer

Megjegyzés: Az összes koordinátát abszolút koordinátával kell megadni, és nem az ablakhoz képest relatívval. Ha sikeres volt a művelet, akkor pozitív a visszatérési érték, különben 0.

_setcursortype	**conio.h**

Kurzor vezérlése.

```
void far _setcursortype(int cut_t);
```

Paraméter:

cur_t a kurzortípus kiválasztása.

Megjegyzés: Alábbi kurzortípusok léteznek:

Kurzortípus	Leírás
_NOCURSOR	kikapcsolja a kurzort
_SOLIDCURSOR	tömör, téglalap alakú kurzor
_NORMALCURSOR	normál (aláhúzás) típusú kurzor

Csak a Borland C++-ban létezik.

textattr	**conio.h**

Beállítja az előtér és az írás színét.

```
void textattr(int newattr);
```

Megjegyzés: Egyetlen hívással be lehet állítani a háttér és az írás színét. A 8 bites *newattr* paraméter a következő:

7	6	5	4	3	2	1	0
B	b	b	b	f	f	f	f

ffff 4 bit az írás színe
bbb 3 bit a háttér színe
B villódzó állapot jelzője

textbackground — conio.h

Kiválasztja a háttér színét

```
void textbackground(int newcolor);
```

Paraméter:
newcolor háttérszín

Megjegyzés: A *newcolor* megadható (0-7) számértékkel vagy szimbolikus konstanssal. A *newcolor* értéke az alábbi lehet:

Szimbolikus konstansok	Érték	Szín
BLACK	0	fekete
BLUE	1	kék
GREEN	2	zöld
CYAN	3	türkiz
RED	4	piros
MAGENTA	5	lila
BROWN	6	barna
LIGHTGRAY	7	világosszürke

textcolor — conio.h

Kiválasztja a karakter színét

```
void textcolor(int newcolor);
```

Paraméter:
newcolor a karakter színe (0-15)

Megjegyzés: A színkonstansok megnevezését lásd a ***setallpalette*** függvény ismertetéséněk. A szöveget lehet villogtatni a BLINK szimbólikus konstans segítségével.

Példa: Sárgán villogjon a kék háttéren a Hello szöveg:

```
textattr(YELLOW+(BLUE<<4)+BLINK);
cputs("Hello");
```

textmode **conio.h**

A szöveges (text) üzemmódot választja ki.

```
void textmode(int newmode);
```

Paraméter:
newmode a text üzemmód típusa.

Megjegyzés: A lehetséges text üzemmódok azonosítói a következők:

Szimbolikus konstans	Érték	Leírás
LASTMODE	-1	előző text mód
BW40	0	fekete/fehér, 40 oszlop
C40	1	színes, 40 oszlop
BW80	2	fekete/fehér, 80 oszlop
C80	3	színes, 80 oszlop
MONO	7	egyszínű, 80 oszlop

wherex **conio.h**

A kurzor x koordinátaértékével tér vissza, amely az aktuális ablakhoz képest relatív távolságot jelent.

```
int wherex (void);
```

Megjegyzés: Visszatérési érték 1-80 közötti egész szám.

wherey **conio.h**

A kurzor y koordinátaértékével tér vissza, amely az aktuális ablakhoz képest relatív távolságot jelent.

```
int wherey (void);
```

Megjegyzés: Visszatérési érték 1-től 25, 43 vagy 50 közötti egész szám.

window	**conio.h**

Szöveg-ablakot definiál a képernyőn.

```
void window(int left, int top, int right, int bottom);
```

Paraméterek:
left, top az ablak bal felső sarka
right, bottom az ablak jobb alsó sarka

Megjegyzés: A szöveg-ablak minimális mérete 1 oszlop és 1 sor. Alapértelmezés szerint a szöveg-ablak a teljes képernyő a következő koordinátákkal: 80 oszlopos módban 1,1,80,25; míg 40 oszlopos módban: 1,1,40,25.

Hangeffektusok létrehozása

sound	**dos.h**

Megszólaltatja a belső hangszórót az adott frekvencián.

```
void sound(unsigned grequency);
```

Paraméter:
frequency a hang frekvenciája Hz-ben.

Megjegyzés: A hangszóró addig szól, amíg a ***nosound*** függvény nem kerül hívásra.

delay	**dos.h**

Felfüggeszti a program végrehajtását egy adott időtartamra.

```
void delay(unsigned msec);
```

Paraméter:
msec a késleltetési ideje [msec] egységben

nosound — dos.h

Kikapcsolja a belső hangszórót. MINTA

```
void nosound(void);
```

Példa:
A következő program 440 Hz-es hangot (normál zenei A) ad 500 ms-ig:

```
#include <dos.h>
void main()
{
    sound(440);
    delay(500);
    nosound();
}
```

F4. Függvények grafikus üzemmódban

arc	**graphics.h**

x,y középpontú körívet rajzol kezdő és végszög között.

```
void far arc(int x, int y,
             int stangle, int endangle, int radius);
```

Paraméterek:

x.y a körív középpontja
stangle a kezdőszög (fokban)
endangle a végszög (fokban)
radius a sugár.

Megjegyzés: Ha a kezdőszögnek 0 és a végszögnek 360 fokot adunk, akkor az függvény teljes kört rajzol. A szögeket az x tengelytől indulva az óramutató járásával ellentétes irányban kell megadni (0 fok 3 órának, 90 fok 12 órának felel meg).

bar	**graphics.h**

Téglalapot rajzol és befesti az aktuális színnel és mintával.

```
void far bar(int left, int top, int right, int bottom);
```

Paraméterek:

left, top a téglalap bal felső sarka
right, bottom a téglalap jobb alsó sarka.

Megjegyzés: A ***setcolor***, ***setfillstyle*** és a ***setlinestyle*** függvények által korábban beállított színnel és mintával rajzolja, illetve tölti ki a téglalapot.

bar3d	**graphics.h**

Téglatestet rajzol és befesti az aktuális színnel és mintával.

```
void far bar3d(int left, int top, int right,
               int bottom, int depth, int topflag);
```

Paraméterek:

left, top	a téglalap bal felső sarka
right, bottom	a téglalap jobb alsó sarka
depth	ha nem nulla, a téglatest teteje zárt,
topflag	ulla esetén a téglatest tetejére újabb téglatest illeszthető.

Megjegyzés: A korábban definiált színnel és mintával rajzol és fest téglatestet.

circle	**graphics.h**

x,y középpontú kört rajzol.

```
void far circle(int x, int y, int radius);
```

Paraméterek:

x, y	a kör középpontjának koordinátái
radius	a kör sugara

Megjegyzés: A kör rajzolása az aktuális színnel történik. A *linestyle* paraméter nem hatásos az ív, kör, ellipszis és ellipszis ív rajzolásánál, csak a *thickness* paraméter használható.

cleardevice	**graphics.h**

Törli a grafikus képernyőt.

```
void far cleardevice(void);
```

Megjegyzés: Törli a képernyőt beszínezve a háttérszínnel és a kurzort a (0,0) pozícióba mozgatja.

clearviewport	**graphics.h**

Törli az aktuális grafikus ablakot.

```
void far clearviewport(void);
```

Megjegyzés: Törli az aktuális grafikus ablakot és a kurzort a (0,0) pozícióba mozgatja.

closegraph **graphics.h**

Lezárja a grafikus üzemmódot.

```
void far closegraph(void);
```

Megjegyzés: A ***closegraph*** a grafikus üzemmód inicializálása előtti képernyő üzemmódot állítja vissza.

detectgraph **conio.h**

Ellenőrzi a hardvert és meghatározza, hogy milyen grafikus meghajtót és módot lehet használni.

```
void far detectgraph(int far *graphdriver,
                     int far *graphmode);
```

Megjegyzés: Megvizsgálja a grafikus kártyát és kiválasztja azt az üzemmódot, amelyik a legjobb felbontást nyújtja. Ha az adott hardverkonfigurációban a grafikus használata nem lehetséges, ezt a **graphdriver* paraméter, illetve a ***graphresult*** függvény -2-es visszatérési értéke jelzi. A lehetséges grafikus üzemmód konstansok az alábbiak:

Grafikus meghajtó konstansok	Numerikus érték
CURRENT_DRIVER	-1
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

A DETECT érték hatására a grafikus rendszer értékeli a grafikus kártya típusát. Minden más érték esetén az adott grafikus kártya legjobb felbontású üzemmódja kerül kiválasztásra.

drawpoly **graphics.h**

A megadott vonaltípussal és színnel pontsorozatot egyenessel köt össze.

```
void far drawpoly(int numpoints,
                  int far *polypoints);
```

Paraméterek:

numpoint: a koordináták

polypoint mutat az x,y pontpárokat tartalmazó tömbre

Megjegyzés: Egy n pontból álló zárt poligon esetén n+1 koordinátapárt kell megadni, ahol az n+1-edik koordinátapárnak meg kell egyeznie a 0-adikkal.

ellipse **graphics.h**

x,y középpontú ellipszis ívet rajzol kezdő és végszög között.

```
void far ellipse(int x, int y, int stangle,
                 int endagnle, int xradius, int yradius);
```

Paraméterek:

x,y	középpont kezdő koordinátái
stangle	kezdeti szög
endangle	végszög
xradius	vízszintes tengely
yradius	függőleges tengely

Megjegyzés: A szögeket az *x* tengelytől indulva az óramutató járásával ellentétes irányban kell megadni. A 0 fok 3 órának, a 90 fok 12 órának felel meg. Ha 0 kezdő- és 360 fok végszöget adunk, teljes ellipszist kapunk.

fillellipse **graphics.h**

Rajzol és befest egy ellipszist.

```
void far fillellipse(int x, int y,
                     int xradius, int yradius);
```

Paraméterek:

x,y	az ellipszis középpontja
xradius	a vízszintes tengely
yradius	a függőleges tengely

Megjegyzés: Rajzol egy *x,y* középpontú, *xradius* vízszintes és *yradius* függőleges tengelyű ellipszist és befesti az aktuális színnel és mintával.

fillepoly graphics.h

Rajzol és befest egy poligont.

```
void far fillpoly(int numpoints, int far *polypoint);
```

Paraméterek:

numpoints	a poligon pontpárainak száma
polypoints	mutat az *x,y* pontpárokat tartalmazó tömbre

Megjegyzés: Aktuális színnel és vonaltípussal megrajzolja a poligon körvonalát és befesti az aktuális mintával és színnel.

floodfill graphics.h

Aktuális mintával befesti az adott színű vonallal zárt területet.

```
var far floodfill(int x, int y, int border);
```

Paraméterek:

x,y	a zárt terület egy belső pontjának koordinátái
border	szín

getarccoord graphics.h

Megadja az utoljára rajzolt ív kezdő- és végkoordinátáinak értékét.

```
void far getarccoords(struct arccoordstype,
                      far *arccoords);
```

Megjegyzés: Visszatér az *arccoordstype* struktúra típusú **arccoords* változóban elhelyezett értékekkel, ahol

```
struct arccoordstype
{
    int x, y;
    int xstart, ystart, xend, yend;
};
```

Ahol

x,y	a középpont koordinátái
xstart, ystart	az ív kezdőpontjának koordinátái
xend, yend	az ív végpontjának koordinátái.

Ezeknek az adatoknak az ismeretében lehet például egy ellipszis ív végpontjából egy vonalat rajzolni.

getaspectratio **graphics.h**

Visszaadja az aktuális grafikus mód vízszintes/függőleges képarányát.

```
void far getaspectratio (int far *xasp,
                         int far *yasp);
```

Paraméterek:

**xasp*, **yasp*	képarány összetevők (faktorok)

Megjegyzés: Az y arányfaktor (**yasp*) minden grafikus kártya esetén 10000-hez van normálva, kivéve a VGA-t. Az **xasp* (x arányfaktor) kisebb, mint az **yasp*, mivel egy képpont (pixel) magassága és szélessége nem egyforma. VGA esetén négyzet alakú egy pixel, emiatt **xasp* egyenlő **yasp*-vel.

getbkcolor **graphics.h**

Háttér színét adja vissza.

```
int far getbkcolor(void);
```

Megjegyzés: Visszatérési értéke a háttérszín, amely 0-15-ig változhat, ez függ a grafikus kártyától és az aktuális grafikus módtól.

getcolor **graphics.h**

A razjoló színt adja vissza.

```
int far getcolor(void);
```

Megjegyzés: Az utolsó sikeres ***setcolor*** hívás színének értékét adja vissza. A rajzolás színe 0-15-ig változhat, ez függ a grafikus kártyától és az aktuális grafikus módtól.

getdafaultpalette **graphics.h**

A paletta (színskála) értékeit adja vissza.

```
struct pallettetype *far getdefaultpalette(void);
```

Megjegyzés: A *palettetype* típusú struktúra mutató pointert kapunk vissza. A pointer által megcímzett struktúrában kapjuk meg az ***initgraph***-ban definiált színskála értékeket.

getdrivername **graphics.h**

Visszatér egy sztringre mutató pointerrel, melyben a grafikus kártya nevét adja vissza.

```
char *far gerdrivername(void);
```

Megjegyzés: Az ***initgraph*** aktiválása után az aktív grafikus kártya nevével tér vissza.

getfillpattern **graphics.h**

A felhasználó által előzőleg definiált alakzat kitöltő minta azonosító kódját adja vissza

```
void far getfillpattern(char far *pattern);
```

Paraméter:
pattern egy pointer, amely egy 8 bájtos szekvenciára mutat.

Megjegyzés: A ***setfillpattern*** által definiált mintát a ***getfillpattern*** betölti egy 8 bájtos területre, amit a *pattern* címez meg.

getfillsettings — graphics.h

Információt ad az aktuális festőmintáról és színről.

```
void far getfillsettings(struct fillsettingstype
                                    far *fillinfo);
```

Megjegyzés: A ***getfillsettings*** betölti a *fillinfo* pointer változóba a ***fillsettingstype*** struktúrára mutató pointert, amely információt ad az aktuális kitöltő mintáról és színről. A struktúra a graphics.h fájlban az alábbi:

```
struct fillsettingstype
{
    int pattern;    /* az aktuális minta */
    int color;      /* az akutális szín   */
}
```

Kitöltő minták

Név (konstans)	Érték	Leírás
EMPTY_FILL	0	háttérszínnel fest
SOLID_FILL	1	egyenletes, gyenge tónus
LINE_FILL	2	vízszintes vonalas minta
LTSLASH_FILL	3	jobbra dőlt vonalas minta
SLASH_FILL	4	jobbra dőlt vastag vonalas minta
BKSLASH_FILL	5	balra dőlt vastag vonalas minta
LTBSKLASH_FILL	6	balra dőlt vonalas minta
HATCH_FILL	7	kockás minta
XHATCH_FILL	8	dőlt kockás minta
INTERLEAVE_FILL	9	sűrűn pontozott minta
WIDE_DOT_FILL	10	ritkán pontozott
CLOSE_DOT_FILL	11	közepesen pontozott
USER_FILL	12	felhasználó által definiált

getgraphmode — graphics.h

Az aktuális grafikus üzemmódot adja meg.

```
int far getgraphmode(void);
```

Megjegyzés: A ***getgraphmode*** megadja az ***initgraph*** vagy ***setgraphmode*** által beállított grafikus üzemmódot. Ennek értéke 0-5 között változhat, ez függ az aktuális grafikus kártyától.

getimage graphics.h

A megadott képmezőt elmenti egy bufferba.

```
void far getimage (int left, int top, int right,
                   int bottom, void far *bitmap);
```

Paraméterek:

left, top	a tartomány bal felső sarka
right, bottom	a tartomány alsó sarka
**bitmap*	a bufferre mutató pointer

Megjegyzés: A ***getimage*** a képmező megadott területét a *bitmap* pointer által mutatott memória területre. A memória területnek az első két szám a tartomány szélességét és magasságát tartalmazza, ezután következnek a képmező adatai, így 4 bájttal nagyobb hely kell a memóriában a képmező tárolására.

getlinesettings graphics.h

A vonal típusát, mintáját és vastagságát adja vissza.

```
void far getlinesettings (struct linesettingstype
                          far *lineinfo);
```

Megjegyzés: A ***getlinesettings*** betölti a *linesettingstype* struktúra pointerét a *lineinfo* pointer változóba, amely a vonal tulajdonságait szolgáltatja:

```
struct linesettingstype
{
    int linestyle;
    unsigned  upattern;
    int thickness;
};
```

A vonal típusa:

Konstans	Érték	Leírás
SOLID_LINE	0	normál vonal
DOTTED_LINE	1	pont vonal
CENTER_LINE	2	közép vonal
DASHED_LINE	3	szaggatott vonal
USERBIT_LINE	4	felhasználó által definiált vonal

A vonal vastagsága

Konstans	Érték	Leírás
NORM_WIDTH	0	normál vonal (1 pixel széles)
THICK_WIDTH	3	vastag vonal (3 pixel széles)

getmaxcolor **graphics.h**

Megadja a maximálisan használható színek számát.

int far getmaxcolor(**void**);

Megjegyzés: Például 256 K-s EGA kártya esetén a visszatérési érték 15, mert a ***setcolor*** maximálisan 0 - 15 érvényes színt tud kiválasztani. A CGA finom grafika és a Hercules egyszínű grafika esetén a visszatérési érték 1, mert ebben az esetben a szín 0 és 1 lehet.

getmaxmode **graphics.h**

A legmagasabb grafikus üzemmód számát adja meg.

```
int far getmaxmode(void);
```

Megjegyzés: A legkisebb érték 0.

getmaxx **graphics.h**

A maximálisan használható x koordináta értéket adja meg.

```
int far getmaxx(void);
```

Megjegyzés: Például a VGA kártya $640 \cdot 480$ felbontású grafikus üzemmódja esetén a ***getmaxx*** 639 értékkel tér vissza.

getmaxy **graphics.h**

A maximálisan használható y koordináta értéket adja meg.

```
int far getmaxy(void);
```

Megjegyzés: Például VGA kártya $640 \cdot 480$ felbontású grafikus üzemmódja esetén a ***getmaxy*** 479 értékkel tér vissza.

getmodename **graphics.h**

A grafikus eszköz meghajtó nevére mutató pointerrel tér vissza.

```
char *far getmodename(int mode_number);
```

Paraméter:
mode_number grafikus mód száma

getmoderange **graphics.h**

Megadja a grafikus meghajtó üzemmódjának tartományát.

```
void far getmoderange(int graphdriver,
                      int far *lomode, int far *himode);
```

Paraméterek:
graphdriver grafikusmeghajtó
lomode, himode az üzemmód tartomány alsó és felső határára mutató pointerek

Megjegyzés: **lomode* tartalmazza a grafikus mód alsó, **himode* pedig a felső határát. Érvénytelen grafikus meghajtó megadása esetén mindkét visszatérési érték -1 lesz.

getpalette **graphics.h**

Információt ad a palettáról.

```
void far getpalette(struct palettetype far *palette);
```

Megjegyzések: A ***getpalette*** feltölti a *palettetype* típusú *palette* struktúrát az aktuálisan használt paletta jellemzőivel. A *palettetype* a graphics.h-ban van definiálva

```
#define MAXCOLOR 15
struct palettetype
{
   unsigned char size;
   signed char colors[MAXCOLORS+1];
};
```

A *size* a színkészletben használható színek száma, melyet az aktuális grafikus mód határoz meg.

getpalettesize **graphics.h**

A színtábla (paletta) méretét adja meg.

```
int far getpalettesize(void);
```

Megjegyzés: A visszatérési érték megadja az aktuális grafikus módban használható színek számát. Például színes üzemmódban használt EGA kártya esetén ez az érték 16 lesz.

getpixel **graphics.h**

Az (x,y) koordinátájú képpont színértékét adja vissza.

```
unsigned far getpixel(int x, int y);
```

Paraméterek:
x, y a képpont koordinátái

gettextsettings **graphics.h**

Információt ad a beállított írásképről.

```
void far gettextsettings(struct textsettingstype
                                    far *texttypeinfo);
```

Megjegyzés: A ***gettextsettings*** betölti a *textsettingstype* struktúra címét a *texttypeinfo* pointer változóba. Ez a struktúra tájékoztatást ad a használt karakterkészlet típusáról, irányáról, méretéről és beállítási helyzetéről.

```
struct textsettingstype
{
  int font;        /* karakterkészlet tipusa    */
  int direction;   /* irány                     */
  int charsize;    /* méret                     */
  int horiz;       /* vízszintes helyzet        */
  int vert;        /* függőleges helyzet        */
};
```

getviewsettings — graphics.h

Az aktuális ablak (viewport) adatait adja meg.

```
void far getviewsettings(struct viewporttype
                               far *viewport);
```

Megjegyzés: A **getviewsettings** kitölti a *viewport* pointer által mutatott *viewporttype* struktúrát.

```
struct viewporttype
{
    int left,top,right,bottom;
    int clip;
};
```

A (*left, top*) és (*right, bottom*) koordinátapontok az aktív ablak méretét szolgáltatják, melyek abszolút képernyő koordinátákban értendők. A *clip* mező értéke határozza meg az ablakban megjelenő rajz vágását. Ha *clip* értéke pozítív, akkor van vágás, ebben az esetben a rajzoknak csak az ablakba eső része látható, nulla esetén nincs vágás.

getx — graphics.h

Az aktuális grafikus kurzor x koordinátáját adja meg.

```
int far getx(void);
```

Megjegyzés: A megadott x koordináta relatív érték az ablakhoz képest, ami azt jelenti, hogy ha a képernyőn egy ablakot jelöltünk ki, akkor az ahhoz viszonyított koordinátákat kapjuk vissza.

gety **graphics.h**

Az aktuális grafikus kurzor y koordinátáját adja meg.

```
int far gety(void);
```

Megjegyzés: A megadott y koordináta relatív érték az ablakhoz képest.

graphdefaults **graphics.h**

Alapállapotba állítja vissza a grafikus üzemmódot.

```
void far graphdefaults(void);
```

Megjegyzés: Az alapállapot beállítás a következő:

- az ablak a teljes képernyőt jelenti,
- a kurzort a (0,0) pozícióba helyezi,
- beállítja a paletta színét, a háttérszínt és a rajzszínt az alapértelmezés szerint, valamint a festő típust, mintát és a szöveginformációt.

grapherrormsg **graphics.h**

Visszatér a hibaüzenetet tartalmazó sztringre mutató pointerrel.

```
char *far grapherrormsg(int errorcode);
```

Megjegyzés: A hibakódot a ***graphresult*** függvény szolgáltatja.

graphfreemem **graphics.h**

A grafikus memória felszabadítása.

```
void far _graphfreemem(void far *ptr, unsigned size);
```

Paraméterek:

**ptr*	a grafikus memória területre mutató pointer
size	a felszabadítandó memória mérete

Megjegyzés: A grafikus könyvtár hívja a ***_graphfreemem*** függvényt, hogy felszabadítsa ***_graphgetmem*** által korábban lefoglalt memóriát. Magunk is vezérelhetjük a grafikus könyvtár memória kezelését egyszerűen, ha definiáljuk a

_graphfreemem függvény saját verzióját. Ennek a rutinnak az alapértelmezés szerinti verziója csak a ***free*** függvényt hívja.

_graphgetmem	**graphics.h**

A grafikus memória lefoglalása.

```
void far *far _graphgetmem(unsigned size);
```

Paraméter:
size a felszabadítandó memória mérete

Megjegyzés: A grafikus könyvtár (nem a felhasználói program) hívja a ***_graphgetmem*** függvényt, hogy memóriaterületet foglaljon le belső bufferek, grafikus meghajtók és karakterkészletek számára. Magunk is vezérelhetjük a grafikus könyvtár memória kezelését egyszerűen, ha definiáljuk a ***_graphgetmem*** függvény saját verzióját. Ennek a rutinnak az alapértelmezés szerinti verziója csak a ***malloc*** függvényt.

graphresult	**graphics.h**

Az utoljára végrehajtott grafikus művelet hibakódját adja meg.

```
int far graphresult(void);
```

Megjegyzés: A hibakód táblázata a 4.1.2 szakaszban található.

imagesize	**graphics.h**

Adott téglalap alakú tartomány méretét adja meg bájtokban.

```
unsigned far imagesize(int left, int top,
                       int right, int bottom);
```

Paraméterek:
left, top a téglalap ba1 felső sarka
right, bottom a téglalap jobb alsó sarka

Megjegyzés: Ha. nagyobb memória szükséges a tárolásra, mint 64 Kbyte, akkor az ***imagesize*** függvény -1 értékkel tér vissza.

initgraph graphics.h

Inicializálja a grafikus rendszert.

```
void tar initgraph( int far *graphdriver,
                    int far *graphmode,
                    char far *pathtodriver);
```

Paraméterek:
graphdriver a grafikus kártya típusa
graphmode grafikus mód
pathtodiver az aktuális .bgi fájlt tartalmazó hozzáférési út (*path*)

Megjegyzés: A ***graphdriver*** paraméternél a DETECT érték megadása esetén a grafikus kártya típusa és a grafikus mód automatikusan kerül kiválasztásra. Ha az aktuális .bgi fájl az aktív DOS könyvtárban van, akkor a *pathtodriver* paraméter értéke "" (üres sztring) lehet.

Grafikus meghajtó konstansok	Értéke
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

Az ***Initgraph*** hívásával a **graphgdriver* az aktuális grafikus meghajtóra és a **graphmode* pedig az aktuális grafikus módra lesz beállítva.

Grafikus módok

Grafkus meghajtó	Grafikus mód	Érték	Oszlopxsor	Paletta	Page (oldal)
CGA	CGAC0	0	320x200	C0	1
	CGAC1	1	320x200	C1	1
	CGAC2	2	320x200	C2	1
	CGAC3	3	320x200	C3	1
	CGAHI	4	640x200	2 szín	1
MCGA	MCGA0	0	320x200	C0	1
	MCGAC1	1	320x200	C1	1
	MCGAC2	2	320x200	C2	1
	MCGAC3	3	320x200	C3	1
	MCGAMED	4	320x200	2 szín	1
	MCGAHI	5	640x480	2 szín	1
EGA	EGALO	0	640x200	16 szín	4
	EGAHI	1	640x350	16 szin	2
EGA64	EGA64LO	0	640x200	16 szín	1
	EGA64HI	1	640x350	4 szín	1
EGAMONO	EGAMONOHI	3	640x350	2 szín	1*
	EGAMONOHI	3	640x350	2 szín	2**
HERC	HERCMONOHI	0	720x348	2 szín	2
ATT400	ATT400C0	0	320x200	C0	1
	ATT400C1	1	320x200	C1	1
	ATT400C2	2	320x200	C2	1
	ATT400C3	3	320x200	C3	1
	ATT400MED	4	640x200	2 szín	1
	ATT400HI	5	640x400	2 szin	1
VGA	VGALO	0	640x200	16 szin	2
	VGAMED	1	640x350	16 szín	2
	VGAHI	2	640x480	16 szín	1
PC3270	PC3270HI	0	720x350	2 szin	1
IBM8514	IBM8514HI	0	1024x768	256 szín	
	IBM8514LO	1	640x480	256 szín	

* 64K az EGAMONO kártyán

** 256K az EGAMONO kártyán

installuserdriver graphics.h

A felhasználó által írt BGI meghajtó installálása a grafikus rendszerbe.

```
int far installuserdriver(char far *name,
                          int huge (*detect)(void));
```

Paraméterek:

name az új meghajtó (.bgi) fájl neve

detect pointer egy szabadon választott függvényre, amely automatikusan érzékeli az új meghajtót.

installuserfont graphics.h

Betölt egy új karakterkészletet, amely nincs beépítve a .bgi rendszerbe.

```
int far installuserfont(char far *name)
```

Paraméter:

name az útvonal neve, ahol az új karakterkészlet van.

Megjegyzés: Egyszerre csak 20 karakter installálható. Hibajelzést kapunk, ha a belső tábla tele van, ilyenkor a függvény a ***grError*** (-11) értékkel tér vissza.

line graphics.h

Egy egyenest rajzol két adott pont között.

```
void far line(int x1, int y1, int x2, int y2);
```

Paraméterek:

xl, yl kezdő koordináta

x2, y2 vég koordináta

Megjegyzés: A két koordinátapont között adott színnel, vonaltípussal és vonalvastagsággal egy egyenest rajzol.

linerel **graphics.h**

Egyenes szakaszt rajzol az aktuális rajzpozíciótól relatív koordinátákkal megadott pontig.

```
void far linerel(int dx, int dy);
```

Paraméterek:

dx	távolság x irányban
dy	távolság y irányban

Megjegyzés: Az egyenest a korábban definiált színnel, vonaltípussal és vonalvastagsággal rajzolja meg.

lineto **graphics.h**

Egyenest rajzol az aktuális rajzpozíciótól az abszolút koordinátákkal adott pontig.

```
void far lineto(int x, int y);
```

Paraméterek:

x, y	az egyenes végpontja

Megjegyzés: Az egyenest a korábban definiált színnel, vonaltípussal és vonalvastagsággal rajzolja meg.

moverel **graphics.h**

Az aktuális rajzpozíciót áthelyezi a relatív koordinátákkal adott helyre.

```
void far moverel(int dx, int dy);
```

Paraméterek:

dx	távolság x irányban
dy	távolság y irányban

moveto **graphics.h**

A rajzpozíciót az abszolút koordinátákkal adott pontba helyezi át.

```
void far moveto(int x, int y);
```

Paraméterek·
x, y az új rajzpozíció abszolút koordinátái

outtext **graphics.h**

Az aktuális rajzpozíciótól kezdve szöveget ír ki.

```
void far outtext(char far *textstring);
```

Paraméter:
textstring a kiírandó szövegre mutató pointer

Megjegyzés: A szöveget a korábban beállított betűtípussal, méretben, valamint a kijelölt irányban (vízszintes, függőleges) írja ki a rajzpozíciótól kezdve.

outtextxy **graphics.h**

Szöveget ír ki a megadott (x,y) ponttól kezdve.

```
void far outtextzy(int x, int y, char far *textstring);
```

Paraméterek:
x, y az adott pont
textstring a kiírandó szöveg

Megjegyzés: A korábban beállított betűtípussal, méretben, valamint a kijelölt irányban (vízszintes, függőleges) szöveget ír ki az adott (*x, y*) ponttól kezdve.

pieslice **graphics.h**

Egy körcikket rajzol és fest.

```
void far pieslice(int x, int y, int stangle,
                  int endangle, int radius);
```

Paraméterek:

x, y	középpont koordinátái
stangle	kezdeti szög
endangle	végszög
radius	sugár

Megjegyzés: Befest egy (*x, y*) középpontú, *radius* sugarú, *stangle* kezdőszögű és *endangle* végszögű körcikket a korábban definiált színnel és mintával. A kezdő- és végszöget x tengelytől kezdve az óramutató járásával ellentétes irányban kell megadni, ahol a 0 fok 3 óránál van, a 90 fok pedig 12 óránál van.

putimage — graphics.h

Korábban tárolt képmező ráhelyezése a képernyőre.

```
void far putimage( int left, int top,
                   void far *bitmap, int op);
```

Paraméterek:

left, top	a képernyőn a téglalap alakú tartomány bal felső sarokpontja
bitmap	pointer, amely a képmezőt tartalmazó területre mutat
op	bináris művelet a kihelyezendő tartomány pontjai és a képernyő pontjai között

Megjegyzés: A tárolt képmező és a képernyő képpontjai között az alábbi bináris műveletek definiálhatók:

Azonosító	Érték	Leírás
COPY_PUT	0	rámásolja
XOR_PUT	1	kizáró *vagy* kapcsolat
OR_PUT	2	*vagy* kapcsolat
AND_PUT	3	*és* kapcsolat
NOT_PUT	4	a képmező inverzét másolja

putpixel — graphics.h

Egy képpontot rajzol az (x,y) pontban.

```
void far putpixel(int x, int y, int color);
```

Paraméterek:

x, y	a pont koordinátái
color	a pont színe

Megjegyzés: Az (x, y) pontban a képpontot az adott színnel rajzolja.

rectangle **graphics.h**

Téglalapot rajzol.

```
void far rectangle(int left, int top,
                   int right, int bottom);
```

Pararnéterek:

left, top	a téglalap bal felső sarka
right, bottom	a téglalap jobb alsó sarka

Megjegyzés: A téglalapot az aktuális színnel és vonaltípussal rajzolja.

registerbgidriver **graphics.h**

```
int registerbgidriver(void (*driver)(void));
```

Megjegyzés: A felhasználó által betöltött vagy a programhoz szerkesztett grafikus meghajtót regisztrálja.

registerbgifont **graphics.h**

```
int registerbgifont(void (*font)(void));
```

Megjegyzés: A felhasználó által betöltött vagy a programhoz szerkesztett karakterkészletet regisztrálja.

restorecrtmode **graphics.h**

Visszaállítja azt a képernyő üzemmódot, amelyik az ***initgraph*** aktiválása előtt volt érvényben.

```
void far restorecrtmode(void);
```

Megjegyzés: A ***restorecrtmode*** visszaállítja az eredeti video módot, amelyet az ***initgraph*** érzékelt. A ***setgraphmode*** függvény visszakapcsolja a grafikus

üzemmódot. A ***textmode*** függvényt csak akkor használhatjuk, ha szöveges üzemmódban van a képernyő és különböző szöveges módokat akarunk váltani.

sector **graphics.h**

Egy ellipszis ívet rajzol és befest.

```
void far sector(int x, int y,
                int stangle, int endangle,
                int xradius, int yradius);
```

Paraméterek:

x, y	középpont koordinátái
stangle	kezdőszög
endangle	végszög
xradius	vízszintes tengely
yradius	függőleges tengely

setactivepage **graphics.h**

Új lapot nyit meg a grafikus output számára.

```
void far setactivepage(int page);
```

Paraméter

page lap száma

Megjegyzés: Több lap használatát csak az EGA (256K), a VGA és a Hercules grafikus kártya teszi lehetővé. A ***setvisualpage*** függvény hívásával váltogathatjuk a látható lapokat, ez segítséget nyújt az animáció számára.

setallpalette **graphics.h**

Változtatja a paletta színeit.

```
void far setallpalette(struct palettetype far *palette);
```

Paraméter:

palette egy *palettetype* típusú struktúrára mutató pointer.

Megjegyzés: EGA/VGA paletta színeket lehet változtatni a ***setallpalette*** függvénnyel. A *palettetype* struktúra a következő:

```
#define MAXCOLOR 15
struct palettetype
{
    unsigned char size;
    signed char colors[MAXCOLORS+1];
};
```

Ahol a tömbben a szín helyén -1 van, ott a paletta színe nem változik.

Lehetséges színek:

CGA		EGA/VGA	
Név	**Szín**	**Név**	**Szín**
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEH	2
CYAN	3	EGA_CYAN	3
RED	2	EGA_RED	2
MAGENTA	5	EGA.MAGENTA	5
BROWN	6	EGA_LIGHTGRAY	7
LIGHTGRAY	7	EGA_BROWN	20
DARRGRAY	6	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_L,IGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEH	58
LIGHTCYAH	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

setaspectratio **graphics.h**

Változtatja a figyelembe veendő vízszintes/függőleges képarányt.

```
void far setaspectratio(int xasp, int yasp);
```

Paraméterek:

xasp, yasp arányténvezők

Megjegyzés: Ha a beépített képaránnyal egy kör torzult, akkor a hibát szoftver úton kiküszöbölhetjük, ha az arányokat változtatjuk.

setbkcolor	**graphics.h**

Beállítja a háttér színét.

```
void far setbkcolor(int color);
```

Paraméter:

color egy szín a palettából, amely lehet egy szám (0-15), vagy a szín szimbolikus neve

Megjegyzés: A *color* paraméterrel az alábbi módon állíthatjuk be kékre a háttér színét

```
setbkcolor(BLUE);
```

A háttér színként az alábbiakat használhatjuk:

Szám	Név	Szám	Név
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
2	CYAH	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGEHTA	13	LIGHTMAGEHTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

setcolor	**graphics.h**

Beállítja a rajzolás színét.

```
void far setcolor(int color);
```

Paraméter

color egy szín a palettából

Megjegyzés: A *color* paraméter értéke 0-tól ***getmaxcolor*** által visszaadott értékéig változhat, amely a rajzolás színét állítja be. EGA esetén 0-tól 15-ig változhat.

A rajzolás színei CGA esetén:

Paletta szám	**1**	**2**	**3**
CGAC0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW
CGAC1	CGA_LIGHTCYAN	CGA_LGHTMAGENTA	CGA_WHITE
CGAC2	CGA_GREEN	CGA_RED	CGA_BROWN
CGAC3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY

A CGA grafikus kártya esetén egyszerre csak négy színt használhatunk, amelyből egy a háttérszín. A 4 grafikus módból azt a módot válasszuk ki, amelynek a rajzolási színértékeit akarjuk felhasználni. Természetesen a módokat váltogathatjuk. Például CGACO módban a paletta 4 színt tartalmaz. Ezek: a háttérszín, halvány zöld, halvány piros és sárga. Ebben a módban a ***setcolor***(CGA_YELLOW) hívás sárgát választja ki rajzolási színnek.

setfillpattern	**graphics.h**

Bitképet definiál a USER_FILL festőminta számára (lásd ***setfillstyle***)

```
void far setfillpattern(char far *upattern, int color);
```

Paraméterek:

upattern	pointer egy 8 byte hosszú memória területre, ez tartalmazza a mintát
color	szín.

setfillstyle	**graphics.h**

Beállítja az aktuális festőmintát és a színt.

```
void far setfillstyle(int pattern, int color);
```

Paraméterek:
pattern minta
color szín

Megjegyzés: A mintaválasztékot lásd a ***getfillsettings*** függvény ismertetésénél.

setgraphbufsize **graphics.h**

Változtatja a belső grafikus buffer méretét.

```
unsigned far setgraphbufsize(unsigned bufsize);
```

Paraméter:
bufsize a buffer mérete

Megjegyzés: A beépített buffer mérete 4096 byte. Ha kisebb is elég, akkor memória területet lehet megtakarítani. Ha a buffer kevésnek bizonyul, -7 hibajelzést kapunk. A beépített buffer mérete egy 650 töréspontú poligon befestéséhez elegendő. Ha ennél több töréspontú poligont akarunk befesteni, akkor meg kell növelni a buffer méretét, hogy elkerüljük a buffer túlcsordulását.

setgraphmode **graphics.h**

Beállítja a grafikus módot és törli a képernyőt.

```
void far setgraphmode(int mode);
```

Paraméter:
mode a beépített grafikus kártya érvényes módja

setlinestyle **graphics.h**

Beállítja a vonal típusát és vastagságát.

```
void far setlinestyle(int linestyle, unsigned upattern,
                      int thickness);
```

Paraméterek:
linestyle vonaltípus
upattern vonaltípus minta
thickness vonalvastagság

Megjegyzés: A *linesettingstype* struktúra a graphics.h fájlban van definiálva:

```
struct linesettingstype
{
  int linestyle;
  unsigned upattern;
  int thickness;
};
```

A *linestyle* paraméter értéke az alábbi lehet:

Név	Érték	Leírás
SOLID_LINE	0	teljes vonal
DOTTED._LINE	1	pontozott vonal
CENTER_LINE	2	középvonal
DASHED_LINE	3	szaggatott vonal
USERBIT_LINE	4	bitmintával megadott

A *thickness* paraméter az alábbi lehet:

Név	Érték	Leírás
NORM_WIDTH	1	normál vastagságú (1 pixel széles)
THICK_WIDTH	3	vastag vonal (3 pixel széles)

setpalette	**graphics.h**

Egy paletta színt változtat.

```
void far setpalette(int colornum, int color);
```

Paraméterek:

colornum szín sorszáma a táblázatban
color szín

Megjegyzés: Ha a *colornum* értéke 0 és a color értéke GREEN, akkor az első elem zöld lesz. A beépített színkonstansok sorszámát lásd a ***setallpalette*** függvény leírásánál.

setrgbpalette **graphics.h**

Felhasználó által definiált színek beállítása IBM8514 típusú meghajtó esetén.

```
void far setrgbpalette(int colornum, int red,
                        int green, int blue);
```

Megjegyzés: A ***setrgbpalette*** az IBM8514 és a VGA meghajtók esetén használható.

settextjustify **graphics.h**

Szöveg helyzetének beállítása az ***outtext*** és az ***outtextxy*** függvények számára.

```
void far settextjustify(int horiz, int vert);
```

Paraméterek:
horiz vízszintes beállítás
vert függőleges beállítás

Megjegyzés: A *horiz* és a *vert* paraméterek az alábbi értékeket vehetik fel:

Név	Érték	Leírás	Paraméter
LEFT_TEXT	0	balra	horiz
CENTER_TEXT	1	középre	horiz
RIGHT_TEXT	2	jobbra	horiz
BOTTOM_TEXT	0	aljára	vert
CENTER_TEXT	1	középre	vert
TOP_TEXT	2	tetjére	vert

settextstyle **graphics.h**

Beállítja az aktuális karakterkészlet típusát és méretét.

```
void far settextstyle(int font, int direction,
                      int charsize);
```

Paraméterek:
font a karakterkészlet neve
direction az írás iránya
charsize karakterek mérete

Megjegyzés: A *font* paraméter az alábbi értékeket veheti fel

Név	Érték	Leírás
DEFAULT_FONT	0	8x8 bitmintájú karakter
TRIPLEX_FONT	1	háromvonalas karakter
SMALL_FONT	3	kisméretű karakter
SANS-SERIF_FONT	4	egyszerű
GOTHIC_FONT	5	gót betű
SCRIPT_FONT	6	írott betű
TRIPLEX_SCR_FONT	7	háromvonalas írott betű
COMPLEX_FONT	8	komplex betű
EUROPEAN_FONT	9	európai betű
BOLD_FONT	10	vastag

A *direction* paraméter az alábbi értékeket veheti fel:

Név	Érték	Leírás
HORIZ_DIR	0	balról jobbra
VERT_DIR	1	alulról felfelé

A *charsize* paraméter lehetséges értékei:

- Ha a *charsize* 1, akkor az ***outtext*** és az ***outtextxy*** a 8x8 bitmintájú karaktereket 8x8 pixeles téglalapban jeleníti meg.
- Ha a *charsize* 2, akkor ezek a kimeneti függvények a 8x8 bitmintájú karaktereket 16x16 pixeles télalapban jelenítik meg és így tovább (10 a maximum)
- Ha a *charsize* 0 akkor a beépített 4-es faktor lesz az érték, vagy a felhasználó által ***setusercharsize*** által definiált méretű lesz.

setusercharsize **graphics.h**

A karakter szélességének és magasságának változtatása.

```
void far setusercharsize(int multx, int divx,
                         int multy, int divy);
```

Paraméterek:

multx divx multx / divx értékkel szorzódik a beépített szélességgel
multy divy multy / divy értékkel szorzódik a beépített magassággal

setviewport **graphics.h**

Ablakot jelöl ki a grafikus képernyőn.

```
void far setviewport(int left, int top, int right,
                     int bottom, int clip);
```

Paraméterek:

left, top az ablak bal felső sarka
right, bottom az ablak jobb alsó sarka
clip pozitív esetén a kivágást bekapcsolja, nulla esetén kikapcsolja

Megjegyzés: A továbbiakban minden koordinátapont az adott ablakhoz lesz viszonyítva. A *clip* paraméter határozza meg, hogy az ablakból kinyúló vonalak látszanak-e.

setvisualpage **graphics.h**

Láthatóvá teszi az adott grafikus ablakot.

```
void far setvisualpage(int page);
```

Pararnéter:

page a lap száma

Megjegyzés: Több lap használata csak EGA (256K), VGA és Hercules grafikus kártya esetén lehetséges.

setwritemode **graphics.h**

Beállítja az írásmódot a vonalrajzolás számára.

```
void far setwritemode(int mode);
```

Paraméter:

mode kétfajta lehet, az alábbi konstansok közül választhatunk:

Szimbólum	Érték	Leírás
COPY_PUT	0	másolás, felülírja a képernyőt
XOR_PUT	1	XOR művelet (kizáró vagy) a képernyővel

Megjegyzés: A COPY_PUT a MOV assembler utasítást használja fel, a vonal felülírja a képernyőt. Az XOR_PUT az XOR utasítást hajtja végre a vonal pontjai és a képernyő pontjai között. Két egymásután következő XOR utasítás a vonalat letörli és a képernyőn az eredeti kép marad meg.

textheight **graphics.h**

Visszatér a szöveg képpontokban mért magasságával.

```
int far textheight(char far *textstring);
```

Paraméter:
textstring szövegre mutató pointer

textwidth **graphics.h**

Visszatér a szöveg képpontokban mért szélességével.

```
int far textvidth(char far *textstring);
```

Paraméter:
textstring szövegre mutató pointer

F5. Borland C++ include fájlok

Az include fájlok, melyeket fejléc (*header*) fájloknak is neveznek, megadják a könyvtári függvények deklarációit, az általuk használt adattípusokat, szimbolikus állandókat, valamint deklarálják a futtató rendszer és a könyvtári függvények által definiált globális változókat. A Borland C++ követi az ANSI C által ajánlott include fájl elnevezéseket és tartalmakat.

A középső oszlop jelzi a C++ fejléc fájlokat és az ANSI C által definiált fejléc fájlokat.

Fejléc fájl		Leírás
alloc.h		Memóriakezelő függvényeket deklarál (tárfoglalás, memória felszabadítás stb.)
assert.h	ANSI C	Definiálja az assert hibakereső makrót.
bcd.h	C++	Deklarálja a bcd C++ osztályt, a hozzátartozó operátorokat és matematikai függvényeket.
bios.h		Az IBM-PC-ROM BIOS rutinok hívásában használt különféle függvényeket deklarál.
bwcc.h		Definiálja a Borland Windows felhasználói vezérlő kapcsolat függvényeit.
checks. h	C++	Definiálja az osztályt ellenőrző makrókat.
complex.h	C++	Deklarálja a complex C++ osztályt, a hozzátartozó operátorokat és matematikai függvényeket.
conio.h		A DOS konzol I/O rutinok hívásában használt különféle függvényeket deklarál.
constrea.h	C++	Definiálja *conbuf* és *constream* osztályokat.
csring.h	C++	Definiálja a *string* osztályt.
ctype.h	ANSI C	Tartalmazza azokat az információkat, melyeket a karakter osztályozó és karakter konverziós makrók használnak (ilyen az pl. ***isalfa*** és ***toascii***).
date.h	C++	A *date* osztályt definiálja.
_defs.h		A különböző alkalmazói típusok és memória modellek számára a definiálja hívási konvenciókat.
dir.h		A könyvtárak és elérési útvonal (path) nevek kezeléséhez tartalmaz struktúra definíciókat, makrókat és függvény - deklarációkat.

direct.h		Definiálja a struktúrákat, makrókat és függvényeket, melyek a könyvtárak és útvonal nevekhez szükséges.
dirent.h		A POSIX könyvtár műveletei számára definiálja a függvényeket és struktúrákat.
dos.h		Szimbólumdefiníciókat és deklarációkat tartalmaz a DOS és 8086 specifikus rendszer hívások számára.
errno.h	ANSI C	Mnemonikus konstansokat definiál a standard hibakódok számára.
except.h	C++	Deklarálja a kivételt kezelő osztályokat és függvényeket.
excpt.h		Deklarálja a C struktúra kivétel támogatását.
fnctl.h		Szimbolikus konstansokat definiál az open könyvtári függvény számára.
file.h	C++	A *file* osztályt definiálja.
float.h	ANSI C	Paramétereket tartalmazza a lebegőpontos rutinok számára.
fstream.h	C++	Deklarálja a C++ fájl I/O-hoz szükséges adatfolyam osztályokat.
generic.h	C++	Makrókat tartalmaz a *generic* osztály deklarációja számára.
io.h		Struktúrákat és deklarációkat tartalmaz az alacsony szintű input/output rutinok számára.
iomanip.h	C++	Deklarálja a C++ I/O manipulátorokat és makrókat tartalmaz paraméterezett manipulátorok létrehozásához.
iostream.h	C++	Deklarálja a C++ alapvető adatfolyam I/O rutinokat.
limits.h	ANSI C	Környezeti információkat, az adott implementációra jellemző korlátokat és sorszámozott típusok értéktartományának határait adja meg konstansok formájában.
locale.h	ANSI C	Deklarálja azokat a függvényeket, amelyek ország- és a nyelvspecifikus információkat adnak.
malloc.h		Deklarálja a memóriakezelő függvényeket és változókat.
math.h	ANSI C	Deklarálja a matematikai függvényeket és a függvényekkel kapcsolatos hibakezelést.
mem.h		Deklarálja a memóriakezelő függvényeket. (Ezek nagy része deklarálva van a string.h állományban is.)
memory.h		Deklarálja a memóriakezelő függvényeket
new.h	C++	A C++ **new** operátorának kezelésével kapcsolatos de-

		finíciókat tartalmaz (*_new_handler* és a *set_new_handler*)
_nfile.h		A nyitott fájlok számának maximumát definiálja.
_null.h		Definiálja a NULL értékét.
process.h		Struktúrákat és deklarációkat tartalmaz a spawn ... és az exec... függvénycsaládok számára.
search.h		Különböző rendező-kereső algoritmusokat megvalósító függvények prototípusát tartalmazza.
setjmp.h	ANSI C	Definiálja a *jmp_buf* típust, amit a ***longjmp*** és a ***setjmp*** függvények használnak, valamint deklarálja ezeket a függvényeket.
share.h		Az osztott fájlkezelést végző függvények számára deklarál paramétereket.
signal.h	ANSI C	Szimbolikus konstansokat és deklarációkat tartalmaz a ***signal*** és a ***raise*** függvények használatához.
stdarg.h	ANSI C	Makrókat definiál a változó számú paraméterrel meghívható függvények argumentumlistáinak kezeléséhez (mint pl. ***vprintf***,***vscanf*** stb.)
stddef.h	ANSI C	Különböző közhasznú adattípusokat ls makrókat definiál.
stdio.h	ANSI C	A Kerninghan és Ritchie által definiált, és a UNIX System V. alatt kiterjesztett szabványos I/O csomag adattípusait és makróit definiálja. Itt találhatók a szabványos, előredefiniált adatfolyam azonosítók (***stdin***, ***stdout***, ***stdprn*** és ***stdderr***) és az adatfolyam jellemző I/O műveleteket végző függvények deklarációja.
stdiostr.h	C++	Deklarálja a C++ (2.0-ás verzió) adatfolyam kezelő osztályokat az stdio FILE struktúrákkal. Az iostream.h is használni kell az új kódban.
stdlib.h	ANSI C	Általános célú függvényeket használ, mint például a konverziós rutinok, a kereső/rendező rutinok stb.
string.h	ANSI C	Deklarálja a különböző string- és memóriakezelő rutinokat.
strstrea.h	C++	Adatfolyam jellegű C++ osztályokat deklarál byte tömbök használatához.
sys/locking.h		Konstansokat definiál a ***locking*** függvény mód paraméteréhez.
sys/stat.h		Fájlok megnyitásához illetve létrehozásához szükséges szimbólumokat definiál.

sys/timeb.h		Deklarálja az ***ftim*** függvényt és az általa visszaadott *timeb* struktúrát.
sys/types.h		Deklarálja a ***time_t*** típust, melyet az időkezelő függvények használnak.
thread.h	C++	Definiálja a többszálat kezelő (*thread*) osztályokat.
time.h	ANSI C	Definiálja azt a struktúrát, amelyet az időkonverziós rutinok (***asctime***, ***localtime*** és ***gtime***) töltenek fel, definiálja azt a típust, amelyet a ***ctime***, ***difftime***, ***gmtime***. ***localtime*** és ***stime*** függvények használnak és egyben deklarálja ezeket a függvényeket.
typeinfo.h	C++	Definiálja a run-time típusú információs osztályt.
utime.h		Deklarálja az ***utime*** függvényt és a *utimbuf* struktúrát, amellyel a függvény visszatér.
values.h		Fontos konstansokat (köztük gépfüggőket) definiál a UNIX System V. operációs rendszerrel való kompatibilitás érdekében.
varargs.h		Makrókat definiál a változó számú paraméterrel meghívható függvények argumentumlistáinak kezeléséhez. Segíti a UNIX kompatibilitást, az stdarg.h használatát javasoljuk.

F6. Általános könyvtári függvények

A következőkben a BORLAND C++ néhány fontosabb, általános célú könyvtári függvényeit ismertetjük. Mindegyik függvény esetében utalunk a portabilitási lehetőségekre is.

abort	**stdlib.h**

Egy program futását fejezteti be ez a függvényt nem hagyományos módon.

```
void abort(void);
```

Megjegyzés: Ha nincs SIGABRT signál kezelés, akkor az *Abnormal program termination* szöveget írja ki a standard hibaperiférián (***stderr***) és aktiválja 3-as kóddal az ***_exit*** függvényt.
Visszatérési érték: 3-as kóddal tér vissza a hívó programba vagy a operációs rendszerbe.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

acos, acosl	**math.h**

Arkusz koszinus értéket számol.

```
double acos(double x);
long double acosl(long double x);
```

Megjegyzés: A **double** argumentumnak -1 és 1 között kell lenni. Hibás argumentum esetén NAN-nal (Not A Number) tér vissza és beállítja az *errno* értékét az *EDOM (jelentése: Domain error)* hibajelzésre. A komplex inverz koszinusz alakja:
Visszatérési érték: 0 és π közötti értékkel tér vissza.
bcd és *complex* típusokra is használhatjuk a függvényt.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
acos	+	+	+	+	+	+	+
acosl	+		+	+			+

asctime — time.h

A dátumot és az időt ASCII karakterlánccá konvertálja.

```
char *asctime(const struct tm *tblock);
```

Megjegyzés: 26 karakteres sztringként adja vissza a **tblock* struktúrában tárolt dátumot és időt (a 26-dik karakter az EOS):
Sun Nov 23 02:05:45 1997\n\0

Visszatérési érték: A karakterláncra mutató pointer. A sztringet a következő ***asctime*** ( ) vagy ***ctime*** ( ) hívás felülírja.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

asin, asinl — math.h

Arkusz szinuszt számol.

```
double asin(double x);
long double asinl(long double x);
```

Megjegyzés: Az *x* argumentumnak -1 és 1 között kell lenni. Hibás argumentum esetén NAN-nal tér vissza és beállítja *errno* értékét az *EDOM* (*Domain error)* hibajelzésre.

Visszatérési érték: $-\frac{\pi}{2}$ és $\frac{\pi}{2}$ közötti értékkel tér vissza.

bcd és *complex* típusokra is használhatjuk a függvényt.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
asin	+	+	+	+	+	+	+
asinl	+		+	+			+

assert — assert.h

Teszteli a feltételt és szükség esetén abortál.

```
void assert(int test);
```

Megjegyzés: Ha a *test* hamis (0), akkor az ***assert*** a következő üzenetet írja ki a ***stderr*** folyamba (a terminálra):

Assertion failed: *test*, file *forrásfile*, line *sor száma*

és az ***abort*** hívása révén terminálja a programot. Ha az assert.h include fájl beépítése előtt definiáljuk az *NDEBUG* (no debugging) szimbólumot (a #define NDEBUG sorral) tetszőleges értékkel, akkor az ***assert*** makró hatástalan lesz.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

atan, atanl — math.h

Arkusz tangest számít.

```
double atan(double x);
long double atanl(long double x);
```

A komplex inverz tangens alakja:

$$\arctan(z) = -0.5 \cdot i \cdot \log \frac{1 + i \cdot z}{1 - i \cdot z}$$

Visszatérési érték: $-\frac{\pi}{2}$ és $\frac{\pi}{2}$ közötti értékkel tér vissza.

bcd és *complex* típusokra is használhatjuk ezt a függvényt.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
atan	+	+	+	+	+	+	+
atanl	+		+	+			+

atan2, atan2l — math.h

y/x arkusz tangensét számítja.

```
double atan2(double y, double x);
long double atan2l(long double y, long double x);
```

Visszatérési érték: $-\frac{\pi}{2}$ és $\frac{\pi}{2}$ közötti értékkel tér vissza.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
atan2	+	+	+	+	+	+	+
atan2l	+		+	+			+

atof, _atold — math.h

Sztringet konvertál lebegőpontos számmá.

```
double atof(const char *s);
long double _atold(const char *s);
```

Visszatérési érték: Az input sztringnek megfelelő lebegőpontos érték. Ha a konvertálás sikertelen, a visszatérési érték 0.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
atof	+	+	+	+	+	+	+
_atold	+		+	+			+

atoi — stdlib.h

Sztringet konvertál rövid egész számmá.

```
int atoi(const char *s);
```

Visszatérési érték: Az input sztringnek megfelelő egész érték. Ha a konvertálás sikertelen, a visszatérési érték 0.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

atol — stdlib.h

Sztringet konvertál hosszú egész számmá.

```
long atol(const char *s);
```

Visszatérési érték: Az input sztring konvertált értéke. Ha a konvertálás sikertelen, a visszatérési érték 0.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

bdos **dos.h**

A DOS rendszer közvetlen hívását végzi.

```
int bdos(int dosfun, unsigned dosdx, unsigned dosal);
```

Megjegyzés: *dosfun* a DOS Reference Manual-ban definiált funkciókód, *dosdx* DX regiszter bemenő értéke, *dosal* AL regiszter bemenő értéke.

Visszatérési érték: A DOS hívás által visszaadott AX regiszterérték.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+				

bsearch **stdlib.h**

Bináris keresés egy rendezett tömbben.

```
void *bsearch(const void *key, const void *base ,
              size_t nelem, size_t width,
              int(_USERENTRY*fcmp)
                 (const void *, const void *));
```

Megjegyzés: A *size_t* típus előjelnélküli egészként van definiálva.

base	Az adott tömb kezdőcíme
key	a keresendő értékre mutató pointer
nelem	a tömb elemeinek száma
width	a tömbelemek mérete ***sizeof*** egységben
fcomp	pointer az összehasonlító függvényre

Az összehasonlító függvényt a ***bsearch*** két pointerrel hívja meg, amelyek a kulcsra, illetve egy bizonyos elemre mutatnak. A függvénynek a következő értékeket kell szolgáltatnia:

- < 0, ha az első pointer mutatta argumentum kisebb a másodiknál
- = 0, ha a két elem megegyezik
- > 0, ha az első pointer mutatta argumentum nagyobb a másodiknál

Visszatérési érték: A megtalált táblabeli elem címe, illetve 0, ha nem talált.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

cabs, cabsl — math.h

Komplex szám abszolút értéke.

double cabs(**struct** complex z);
long double cabsl(**struct** complex z);

Megjegyzés: A struktúra az alábbi:

```
struct complex {
   double x, y;
}
```

ahol az *x* a valós és *y* a képzetes rész.

Visszatérési érték: A *z* komplex szám abszolút értéke vagy HUGE_VAL túlcsordulás esetén.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
cabs	+	+	+	+	+	+	+
cabsl	+		+	+			+

calloc — stdlib.h

Memóriaterületet foglal le (allokál).

```
void *calloc(size_t nitems, size_t size);
```

Megjegyzés: A ***calloc*** lefoglal egy *nitems*size* méretű memóriaterületet és kinullázza.
Visszatérési érték: A lefoglalt blokkra mutató pointer. Ha a lefoglalandó méretre nincs hely, vagy *nitems* vagy *size* 0, akkor NULL-t ad vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

chdir **dir.h**

Aktuális katalógus (directory) váltása.

```
int chdir(const char *path);
```

Megjegyzés: Lemezegység is specifikálható a *path* argumentumban, például

```
chdir("a:\\borlandc");
```

Visszatérési érték: Sikeres végrehajtás esetén a visszatérési érték 0, különben -1 és az *errno* a következő hibajelzésre lesz beállítva: ENOENT a *path* (útvonal) vagy a fájl név nem létezik.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

clock **time.h**

Processzor idő lekérdezése.

```
clock_t clock(void);
```

Megjegyzés: A ***clock*** segítségével meghatározható két esemény között eltelt idő. Ha az értéket másodpercben kívánjuk megkapni, a visszaadott értéket el kell osztani a CLK_TCK szimbólummal.

Visszatérési érték: A program indulása óta eltelt processzor idő belső egységben.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+	+	+	+

close **io.h**

Lezár egy fájlt.

```
int close(int handle);
```

Megjegyzés: A *handle* fájl leíróval azonosított fájlt zárja le. Nem ír Ctrl-Z karaktert a fájl végére. Ha Ctrl-Z-vel akarjuk a fájlt lezárni, akkor azt explicit módon kell odatenni.

Visszatérési érték: Hiba esetén az *errno* változót beállítja a EBADF értékre (rossz fájl azonosító szám).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

cos, cosl — math.h

Koszinusz értéket számol.

```
double cos(double x);
long double cosl(long double x);
```

Megjegyzés: *bcd* és *complex* típusokra is hívható a függvény.
Visszatérési érték: A radiánban megadott szög koszinusza (-1 és 1 közötti érték).

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
cos	+	+	+	+	+	+	+
cosl	+		+	+			+

cosh, coshl — math.h

Koszinusz hiberbolikusz értéket számol.

```
double cosh(double x);
long double coshl(long double x);
```

Megjegyzés: *bcd* és *complex* típusokra is hívható a függvény.
Visszatérési érték: Az argumentum koszinus hiperbolikusza.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
cosh	+	+	+	+	+	+	+
coshl	+		+	+			+

creat — io.h

Új fájlt hoz létre, vagy felülír egy létező fájlt.

```
int creat(const char *path, int amode);
```

Megjegyzés: Létrehozza vagy felülírja a *path*-ban megadott nevű fájlt *amode* hozzáférési móddal.

amode változó értéke hozzáférési mód

***amode* értéke**	**Hozzáférési mód**
S_IWRITE	csak írásra
S_IREAD	csak olvasásra
S_IREAD / S_IWRITE	írásra és olvasásra

Visszatérési érték: A fájleíró, vagy hiba esetén -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

ctime — time.h

A dátumot és az időt ASCII karakterlánccá konvertálja.

```
char *ctime(const time_t *time);
```

Megjegyzés: **time* egy megelőző ***time***() hívással állítható be. A dátumot és az időt az alábbi formájú, újsor és EOS karakterrel lezárt 26 karakteres sztringben szolgáltatja:

Sun Nov 23 01:08:45 1997\n\0

Visszatérési érték: A karakterláncra mutató pointer. A sztringet a következő ***asctime***(), vagy ***ctime***() hívás felülírja.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ecvt — stdlib.h

Lebegőpontos számot sztringbe konvertál.

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Megjegyzés: A *value* értékét konvertálja *ndig* számjegyet tartalmazó sztringgé, **dec*-be a tizedespont pozícióját teszi a rutin (maga a tizedespont nem szerepel a sztringben), **sign*-ba pedig nem 0 kerül, ha az érték negatív.

Visszatérési érték: A sztringre mutató pointer, ami egy statikus bufferben van. ***ecvt*** következő hívása felül fogja írni az új értékkel.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

eof — io.h

Fájlvég ellenőrzése.

```
int eof(int handle);
```

Visszatérési érték: Ha az aktuális pozíció a fájlvég, az *eof* visszatérési értéke 1, különben 0.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

exit — stdlib.h

Befejezi a program futását és visszaadja a vezérlést az operációs rendszernek vagy a hívó programnak.

```
void exit(int status);
```

Megjegyzés: A terminálás előtt az összes fájlt lezárja. A visszaadott *status*-nál használható szimbólumok:

EXIT_SUCCESS	hibátlan programbefejezés
EXIT_FAILURE	befejezés hibával.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

_exit — stdlib.h

Befejezi a program futását.

```
void _exit(int status);
```

Megjegyzés: A fájlok lezárása nélkül fejezi be a végrehajtást.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

exp, expl — math.h

e^x értéket számol.

```
double exp(double x);
long double expl(long double x);
```

Megjegyzés: *bcd* és *complex* típusok is használhatják a függvényt.
Visszatérési érték: e^x

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
exp	+	+	+	+	+	+	+
expl	+		+	+			+

farcalloc — alloc.h

Memóriát foglal le az alapszegmensen kívül.

```
void far *farcalloc(unsigned long nunits,
                    unsigned long unitsz);
```

Megjegyzés: Ugyanaz a funkciója, mint a *calloc*-nak, de *compact, large* és *huge* modellből is lehetővé teszi az összes rendelkezésre álló RAM lefoglalását, valamint segítségével 64 Kbyte-nál nagyobb tömbök számára is foglalhatunk le memóriát. A *tiny* kivételével minden modellből hívható.

Visszatérési érték: Egy **far** pointer, mely az újonnan lefoglalt blokkra mutat, illetve NULL, ha nincs elég memória.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+				

farfree	**alloc.h**

Felszabadít egy memóriablokkot.

```
void farfree(void far *block);
```

Megjegyzés: block egy megelőző ***farmalloc***(), ***farcalloc***() illetve ***farrealloc***() hívásból származhat.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+				

farmalloc	**alloc.h**

Memóriát foglal le az alapszegmensen kívül.

```
void far *farmalloc(unsigned long nbytes);
```

Megjegyzés: Lásd ***malloc***-ot és ***farcalloc***-ot.

Visszatérési érték: Egy **far** pointer, mely az újonnan lefoglalt blokkra mutat, illetve NULL, ha nincs elég memória.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+				

farrealloc — alloc.h

Módosítja egy lefoglalt memóriaterület méretét.

```
void far *farrealloc(void far *oldblock,
                     unsigned long nbytes);
```

Megjegyzés: *nbytes* nagyságúra módosítja az *oldblock* mutatta, megelőző ***farmalloc***() vagy ***farcalloc***() hívásból származó allokált blokkot. Tartalmát új területre másolja, ha a kérés teljesítése az eredeti helyen nem lehetséges.

Visszatérési érték: Az újra allokált blokk címe, amely lehet, hogy eltér *oldblock*-tól. Ha a kért méretnövelés nem teljesíthető, NULL értékkel tér vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+				

fclose — stdio.h

Fájlt zár le.

```
int fclose(FILE *stream);
```

Megjegyzés: A *stream* fájlmutatóval azonosított fájlt zárja le.

Visszatérési érték: A visszatérési érték sikeres lezárás esetén 0, különben hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

feof — stdio.h

Fájlvég pozíció detektálása.

```
int eof(FILE *stream);
```

Visszatérési érték: A visszatérési érték nem 0, ha a stream fájlmutatóval azonosított file aktuális pozíciója a fájlvég.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ferror stdio.h

Átviteli hiba lekérdezése.

```
int ferror(FILE *stream);
```

Megjegyzés: A ***getc*** stb. rutinok mind fájl vége esetén, mind adatátviteli hiba esetén EOF értékkel térnek vissza. A ***ferror*** és a ***feof*** használható annak eldöntésére, hogy melyik eset állt elő.

Visszatérési érték: A visszatérési érték nem 0, ha a *stream* fájlmutatóval azonosított fájl írása vagy olvasása közben adatátviteli hiba volt.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fgets stdio.h

Egy sor beolvasása adott fájlból.

```
char *fgets(char *s, int n, FILE *stream);
```

Megjegyzés: A *stream* által azonosított fájlból maximum *n-1* karaktert az *s* sztringbe olvas, de leáll az olvasás az első beolvasott újsor karakter után. A sztring végére kiteszi az EOS karaktert.

Visszatérési érték: Sikeres olvasás esetén az *s* sztringre mutató pointer, hiba esetén NULL.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fopen stdio.h

Megnyit vagy létrehoz egy fájlt folyam jellegű kezelésre.

```
FILE *fopen(const char *filename, const char *mode);
```

Megjegyzés: Megnyitja vagy létrehozza a *filename* nevű fájlt a *mode* értékének megfelelően. A név tartalmazhat meghajtó (*drive*) és útvonal (*path*) megadást is, de ne felejtsük el a '\' karaktereket megkettőzni. A *mode* sztring értékei az alábbiak lehetnek:

Érték	Leírás
r	Létező fájl megnyitása csak olvasásra.
w	Új fájl létrehozása (vagy létező felülírása) és megnyitása csak írásra.
a	Létező fájl megnyitása hozzáfűzésre (*append*), vagy új fájl létrehozása csak írásra, ha nem létezik.
r+	Létező fájl megnyitása olvasásra és írásra.
w+	Új fájl létrehozása (vagy létező felülírása) és megnyitása olvasásra és írásra.
a+	Megnyitás hozzáfűzésre. Ha a fájl nem létezik, először létrehozza.

Azoknál a kezelési módoknál, ahol írás és olvasás egyaránt lehetséges, átviteli irány váltás esetén meg kell hívni az ***fseek***, vagy ***rewind*** függvényeket a belső pufferek ürítése céljából. A *mode* sztring minden fent megadott értékéhez hozzátoldhatjuk a 't' vagy 'b' karaktert annak jelzésére, hogy szöveges (text) vagy bináris módban kívánjuk a fájlt kezelni. Ha nem adjuk meg egyiket sem, akkor a fájl az *_fmode* nevű globális változó által tárolt mód szerint lesz megnyitva. Az *_fmode*-nak értékül az fcntl.h include fájlban definiált O_TEXT vagy O_BINARY szimbólumot adhatjuk.

Visszatérési érték: Sikeres megnyitás esetén a fájlmutató, hiba esetén a NULL pointer.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fprintf stdio.h

Formázott kivitel fájlba.

```
int fprintf(FILE *stream,
            const char *format[,argument,...]);
```

Megjegyzés: Működése megegyezik a ***printf*** rutinéval, de megadható a kimeneti folyam a *stream* fájlmutatóval. További részleteket a ***printf*** függvénycsalád ismertetésénél, a 1.8.-as részben találhatunk.

Visszatérési érték: A kiírt bájtok száma, vagy hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fputs stdio.h

Fájlba ír egy karakterláncot.

```
int fputs(const char *s, FILE *stream);
```

Megjegyzés: Kiírja az *s* sztringet a *stream*-mel azonosított fájlba. A záró EOS nem kerül kivitelre.
Visszatérési érték: Sikeres végrehajtás esetén az utolsó kiírt karakterrel tér vissza, egyébként a visszatérési érték EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fread stdio

Fájlból tömböt olvas.

```
size_t fread(void *ptr, size_t size, size_t n,
             FILE *stream);
```

Megjegyzés: A *stream*-mel azonosított fájlból beolvas *n* darab *size* méretű adatot a *ptr* által mutatott tömbbe.

Visszatérési érték: A beolvasott adatok (nem a bájtok) száma. Hiba, vagy fájlvég esetén *n*-nél kevesebbet ad vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

free **stdlib.h**

Felszabadítja az allokált memóriablokkot.

```
void free(void *block);
```

Megjegyzés: Felszabadít egy lefoglalt memóriablokkot. A *block* egy megelőző ***calloc***(), ***malloc***() vagy a ***realloc***() hívás révén kapott mutató.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fscanf **stdio.h**

Fájlból olvas formátum szerint

```
int fscanf(FILE *stream,
           const char *format[,address,...]);
```

Megjegyzés: Működése megegyezik a ***scanf*** rutinéval, de megadható a bemeneti folyam a *stream* fájlmutatóval.
Visszatérési érték: A sikeresen beolvasott mezők száma. Fájlvég esetén EOF-ot ad.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fseek **stdio.h**

File aktuális pozíciójának beállítása

```
int fseek(FILE *stream, long offset, int origin);
```

Megjegyzés: Beállítja a *stream* által azonosított fájlban az aktuális fájlpozíciót az *origin*-hez képest *offset* bájtra.
Az *origin* az alábbi értékeket veheti fel:

Szimbólum	Számérték	offset számítása
SEEK_SET	0	a fájl elejétől
SEEK_CUR	1	az aktuális fájlpozíciótól
SEEK_END	2	a fájl végétől

Visszatérési érték: Sikeres végrehajtás esetén 0, különben más érték.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ftell **stdio.h**

Az aktuális fájlpozíciót szolgáltatja.

```
long int ftell(FILE *stream);
```

Megjegyzés: A fájlpozíció a fájl elejétől bájtokban kifejezett távolság, 0L-ról indul.
Visszatérési érték: Az aktuális fájlpozíció értéke sikeres végrehajtás esetén, egyébként -1L.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

fwrite **stdio.h**

Fájlba tömböt ír.

```
size_t fwrite(const void *ptr, size_t size,
              size_t n, FILE *stream);
```

Megjegyzés: A *stream*-mel azonosított fájlba kiír *n* darab *size* méretű adatot a *ptr* mutatta tömbből.

Visszatérési érték: A kiírt adatok (nem a bájtok) száma. Hiba esetén *n*-nél kevesebbet ad vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

gcvt — stdlib.h

Lebegőpontos szám ASCII karakterlánccá konvertálása.

```
char *gcvt(double value, int ndec, char *buf);
```

Megjegyzés: *value* értékét konvertálja ndec értékes számjegyre fixpontos alakban (FORTRAN F forma), ha lehet, egyébként lebegőpontos alakban (FORTRAN E forma). Az eredmény a *buf* mutatta pufferbe kerül, EOS karakterrel lezárva.
Visszatérési érték: *buf.*

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

getc — stdio.h

Egy karaktert olvas a fájlból.

```
int getc(FILE *stream);
```

Megjegyzés: Beolvassa a fájl következő karakterét, és egész számmá konvertálja előjelkiterjesztés nélkül.
Visszatérési érték: A beolvasott karakter, illetve fájlvég vagy hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

getch — conio.h

Karaktert olvas a klaviatúráról, képernyőre írás (echo) nélkül.

```
int getch(void);
```

Megjegyzés: A ***getch*** a *stdin* folyamot (standard input) használja.
Visszatéiési érték: a beolvasott karakter.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+			+			+

getchar — stdio.h

Karaktert olvas a *standard* inputról.

```
int getchar(void);
```

Megjegyzés: Megegyezik a ***getc***(*stdin*) hívással.
Visszatérési érték: a beolvasott karakter.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

getche — conio.h

Karaktert olvas a klaviatúráról és visszaírja a képernyőre.

```
int getche(void);
```

Visszatérési érték: a beolvasott karakter.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+			+			+

gets stdio.h

Beolvas egy sort a standard inputról.

```
char *gets(char *s);
```

Megjegyzés: A beolvasott sort az *s* sztring változóba helyezi, az újsor karaktert EOS karakterrel helyettesítve.
Visszatérési érték: Sikeres olvasás esetén *s*, fájlvég vagy hiba esetén NULL.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

gmtime time.h

A time_t típusban megadott dátumot és időt struktúrába bontja.

```
struct tm *gmtime(const time_t *time);
```

Megjegyzés: **time* feltöltését a ***time()*** hívásával végezhetjük. A *tm* struktúra definíciója a *time.h* fejléc fájlban található.
Visszatérési érték: A struktúrára mutató pointer. A következő ***gmtime***() hívás felülírja az új értékkel.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isalnum ctype.h

Karakter osztályozó makró.

```
int isalnum(int c);
```

Visszatérési érték: nem nulla, ha a c betű (A-Z, a-z) vagy számjegy (0-9).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isalpha ctype.h

Karakter osztályozó makró.

```
int isalpha(int c);
```

Visszatérési érték: nem nulla, ha a c betű (A-Z vagy a-z).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

iscntrl ctype.h

Karakter osztályozó makró.

```
int iscntrl(int c);
```

Visszatérési érték: nem nulla, ha a *c* értéke a DEL karakter, vagy vezérlőkarakter (0x7F, vagy 0x00 - 0x1F).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isdigit ctype.h

Karakter osztályozó makró.

```
int isdigit(int c);
```

Visszatérési érték: nem nulla, ha a *c* értéke számjegy: (0 - 9).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

islower — ctype.h

Karakter osztályozó makró.

```
int islower(int c);
```

Visszatérési érték: nem nulla, ha a *c* kisbetű (a - z);

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isprint — ctype.h

Karakter osztályozó makró.

```
int isprint(int c);
```

Visszatérési érték: nem nulla, ha a *c* nyomtatható karakter.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isspace — ctype.h

Karakter osztályozó makró.

```
int isspace(int c);
```

Visszatérési érték: nem nulla, ha a *c* white space karakter, azaz betűköz (*space*), tabulátor, kocsivissza (*carrige return*), újsor, vizszintes vagy függőleges tabulátor, vagy lapdobás karakter (0x09 - 0x0D, 0x20).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isupper — ctype.h

Karakter osztályozó makró.

```
int isupper(int c);
```

Visszatérési érték: nem nulla, ha a *c* nagybetű (A - Z).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

isxdigit — ctype.h

Karakter osztályozó makró.

```
int isxdigit(int c);
```

Visszatérési érték: nem nulla, ha a *c* hexadecimális számjegy (0 - 9, A-F, a - f).

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

itoa — stdlib.h

Egész számot sztringgé konvertál.

```
char *itoa(int value, char *sztring, int radix);
```

Megjegyzés: A *value* értékét EOS karakterrel lezárt sztringgé konvertálja a *sztring* mutatta tömbbe. A *radix* a konvertálás alapszámát határozza meg (2-36). Ha a *value* negatív, és *radix* 10, akkor előjelesen konvertál, egyébként előjeltelenül.
Visszatérési érték: a *sztring*-re mutató pointer.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

kbhit conio.h

Megvizsgálja, hogy várakozik-e karakter a billentyű-pufferben.

```
int kbhit(void);
```

Visszatérési érték: Nem nulla, ha van beolvasható karakter.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

lfind stdlib.h

Lineáris keresést hajt végre.

```
void *lfind(const void *key, const void *base,
            size_t *num, size_t width,
            int (_USERENTRY *fcmp)
            (const void *, const void *));
```

Megjegyzés: A **key* értéke szerint lineáris keresés történik a *base* tömbben a felhasználó által definiált ***fcmp*** összehasonlító rutin felhasználásával (lásd ***bsearch***). A tömb **num* elemből áll, *width* az elemek ***sizeof*** mérete.
Visszatérési érték: Az első egyező tömbelem címe illetve NULL, ha nincs ilyen.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

log, logl math.h

Természetes alapú logaritmust számol.

```
double log(double x);
long double logl(long double x);
```

Megjegyzés: bcd és *complex* típusok is használhatják.

Visszatérési érték: ln(x), hibás argumentum esetén EDOM (Domain error) hibajelzést ad.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
log	+	+	+	+	+	+	+
logl	+		+	+			+

log10, log10l math.h

Tizes alapú logaritmust számol.

```
double log10(double x);
long double log10l(long double x);
```

Megjegyzés: *bcd* és a *complex* típusok is használhatják.
Visszatérési érték: lg(x), hibás argumentum esetén EDOM (Domain error) hibajelzést ad.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
log10	+	+	+	+	+	+	+
log10l	+		+	+			+

lsearch stdlib.h

Lineáris keresést hajt végre.

```
void *lsearch(const void *key, const void *base,
              size_t *num, size_t width,
              int (_USERENTRY *fcmp)
              (const void *, const void *));
```

Megjegyzés: A **key* értéke szerint lineáris keresés történik a *base* tömbben a felhasználó által definiált ***fcmp*** összehasonlító rutin felhasználásával (lásd ***bsearch***). A tömb **num* elemből áll, *width* az elemek ***sizeof*** mérete. Ha a keresett elemet nem találja, akkor a tömb végéhez hozzáilleszti (*append*).
Visszatérési érték: Ha új elemet illesztett a tömbbe, **num* értékét módosítja.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

lseek — io.h

Fájlpozíciót állít be alacsony szintű I/O művelethez.

```
long lseek(int handle, long offset, int origin);
```

Megjegyzés: Beállítja a *handle* fájlleíró által azonosított fájlban az aktuális file-pozíciót az *origin*-hez képest *offset* bájtra. Az *origin* az alábbi értékeket veheti fel:

Szimbólum	Számérték	offset számítása
SEEK_SET	0	a fájl elejétől
SEEK_CUR	1	az aktuális fájl pozíciótól
SEEK_END	2	a fájl végétől

Visszatérési érték: Sikeres végrehajtás esetén a beállított fájlpozíció a fájl elejétől számítva, egyébként -1L.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

ltoa — stdlib.h

Hosszú egész számot sztringgé konvertál.

```
char *ltoa(long value, char *sztring, int radix);
```

Megjegyzés: A *value* értékét, EOS karakterrel lezárt sztringgé konvertálja a *sztring* mutatta tömbbe. A *radix* a konvertálás alapszámát határozza meg (2-36). Ha a *value* negatív, és *radix* 10, akkor a sztring első karaktere "-" lesz.
Visszatérési érték: a *sztring*-re mutató pointer.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

malloc — stdlib.h, alloc.h

Dinamikus nemóriafoglalás.

```
void *malloc(size_t size);
```

Megjegyzés: *size* bájtnyi memóriát foglal le futás közben.
Visszatérési érték: az újonnan lefoglalt memóriablokkra mutató pointer, ha a kérés teljesíthető, egyébként NULL.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

open — fcntl.h, io.h

Fájlnyitás alacsony szintű írásra/olvasásra.

```
int open(const char *path, int access
         [,unsigned mode]);
```

Megjegyzés: Megnyitja a *path*-ban specifikált fájlt, és előkészíti írásra és vagy olvasásra a *access* paraméter szerint. A *accsess* az alábbi szimbólumok bináris OR kapcsolatával (|) képezhető. Ebből a csoportból pontosan egy szimbólum szerepelhet:

Read/Write jelzők:	
O_RDONLY	Nyitás csak olvasásra.
O_WRONLY	Nyitás csak írásra.
O_RDWR	Nyitás olvasásra és írásra.

A további felhasználható szimbólumok:

O_APPEND	Megadása esetén minden írási műveletet megelőzően a fájl-pozíció a fájl végére lesz állítva.
O_CREAT	Ha a fájl nem létezik, akkor létre kell hozni.
O_EXCL	Ha a fájl létezik és O_CREAT kérelem volt, hibával tér vissza.
O_TRUNC	Ha a fájl létezik, akkor levágja 0 hosszúságúra.
O_BINARY	A fájlt bináris kezelési módban nyitja meg.
O_TEXT	A fájlt szöveges kezelési módban nyitja meg.

Ha a *mode* argumentum tartalmazza O_CREAT-ot, akkor az *attrib* értékei:

S_IWRITE	engedélyezés írásra
S_IREAD	engedélyezés olvasásra
S_IREAD | S_IWRITE	engedélyezés írásra és olvasásra

Visszatérési érték: A fájl-leíró sikeres végrehajtás esetén, egyébként -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

perror **stdio.h**

Rendszer hibaüzenetet ad.

```
void perror(const char *s);
```

Megjegyzés: Az ***stderr*** perifériára (általában a képernyő) kiírja az s sztringet, egy kettőspontot és az *errno* tartalmának megfelelően a legutoljára bekövetkezett rendszerhiba megnevezését.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

printf **stdio.h**

Formattált kimenet az *stdout*-ra.

```
int printf(const char *format[,argument,...]);
```

Megjegyzés: A *format* formátumsztringnek megfelelően konvertálja az argumentumait, és az így nyert karaktereket az *stdout* perifériára küldi. A formátumsztring kétféle információelemből épül fel: egyszerű karakterekből (ezek változtatás nélkül kerülnek át a kimenetre), illetve konverzió-specifikációkból (ezek a soron következő argumentum megfelelő feldolgozását és kiíratását írják elő). Részletes ismertetőt találhatunk a *printf* függvényről a 1.8.-as részben.
Visszatérési érték: A kiírt bájtok száma, hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

putc — stdio.h

Karaktert ír ki egy folyam-jellegű fájlba.

```
int putc(int c, FILE *stream);
```

Megjegyzés: Makró, amely kiírja a c karaktert a *stream* fájlmutatójú állományba.
Visszatérési érték: A kiírt karakter, hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

putchar — stdio.h

Karaktert ír az *stdout* perifériára.

```
int putchar(int c);
```

Megjegyzés: Makró, amely megfelel a ***putc****(c, stdout*) hívásnak.
Visszatérési érték: A kiírt karakter, hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

puts — stdio.h

Karakterláncot ír ki az stdout perifériára.

```
int puts(const char *s);
```

Megjegyzés: Az s karakterláncot írja ki az *stdout* folyamba, és automatikusan kiegészíti egy újsor karakterrel (' \ n').

Visszatérési érték: Sikeres kiírás esetén pozitív érték, egyébként EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

qsort — stdlib.h

Tömb rendezése gyorsrendező (quicksort) algoritmussal.

```
void qsort(void *base, size_t nelem, size_t width,
           int (_USERENTRY *fcmp)
           (const void *, const void *));
```

Megjegyzés: Tetszőleges elemekből álló tömb rendezését végzi a felhasználó által biztosított összehasonlító függvény segítségével. Az egyes paraméterek:

base	Az adott tömb kezdőcíme
nelem	a tömb elemeinek száma
width	a tömbelemek mérete ***sizeof*** egységben
fcmp	pointer az összehasonlító függvényre

Az összehasonlító függvényt a ***qsort*** két pointerrel hívja meg, amelyek egy-egy elemre mutatnak. A függvénynek a következő értékeket kell szolgáltatnia:

<0,	ha az első pointer mutatta argumentum kisebb a másodiknál
= 0,	ha a két elem megegyezik
>0,	ha az első pointer mutatta argumentum nagyobb a másodiknál.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

rand — stdlib.h

Véletlenszám-generátor.

```
int rand(void);
```

Visszatérési érték: Egy álvéletlen szám 0 és RAND_MAX között.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

random stdlib.h

Véletlenszám-generátor.

```
int random(int num);
```

Visszatérési érték: 0 és (num-1) közötti véletlenszám.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

read io.h

Alacsony szintű fájl olvasás.

```
int read(int handle, void *buf, unsigned len);
```

Megjegyzés: *len* bájtot olvas a *handle*-vel azonosított fájlból *buf*-ba.
Visszatérési érték: A sikeresen beolvasott bájtok száma. Fájlvég esetén 0, hiba esetén -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

realloc stdlib.h

Módosítja a ***malloc***, illetve a ***calloc*** által lefoglalt memóriablokkot.

```
void *realloc(void *block, size_t size);
```

Megjegyzés: A *block* argumentum mutat az előzőleg lefoglalt memóriaterületre, amelyet *size* méretűre kell módosítani. Ha a blokkot növelni kell, akkor szükség esetén a régi blokk tartalmát átmásolja az új helyre.

Visszatérési érték: A memóriaterület címe, amely különbözhet *block*-tól. Ha a kért növelés nem teljesíthető, NULL értékkel tér vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

scanf	**stdio.h**

Olvassa és formázza az *stdin* bemenetet.

```
int scanf(const char *format[,address, ...]);
```

Megjegyzés: karaktereket olvas a szabványos bemenetről és azokat a *format* formátumsztring szerint megpróbálja értelmezni és konvertálás után tárolni. Részletesebb leírást találhatunk a ***scanf*** függvénycsaládról a 1.8.-as részben.
Visszatérési érték: A sikeresen beolvasott és eltárolt tételek száma. Fájlvég olvasása esetén EOF-ot ad vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

setbuf	**stdio.h**

Egy folyamhoz rendelt buffer méretét állítja be.

```
void setbuf(FILE *stream, char *buf);
```

Megjegyzés: közvetlenül a *stream* megnyitása után hívható meg. Az adatok bufferelésére a *buf* memóriaterület használatát írja elő, az automatikusan lefoglalt buffer helyett.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

setmode	**io.h**

A fájl kezelési módját állítja át.

```
int setmode(int handle, int mode);
```

Megjegyzés: a *mode* értéke az alábbi lehet:

O_BINARY bináris
O_TEXT text (szöveg) típusú.

Visszatérési érték: Hiba esetén -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

setvbuf stdio.h

A folyam bufferelését. írja elő.

```
int setvbuf(FILE *stream, char *buf,
            int type, size_t size);
```

Megjegyzés: *size* méretű buffert ír elő a *stream* folyam számára, *type* bufferelési eljárás mellett. Ha *buf* == NULL, akkor automatikusan foglal buffert, egyébként a megadott buffert használja. A *type* értéke paraméter az alábbi lehet:

_IOFBF teljesen pufferelt folyam
_IOLBF sorpufferelt folyam
_IONBF nem pufferelt folyam

Visszatérési érték: nulla, ha sikeres.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+	+	+	+

sin, sinl math.h

Szinusz értéket számol.

```
double sin(double x);
long double sinl(long double x);
```

Megjegyzés: A függvényt *bcd* és *complex* típusok is használhatják.
Visszatérési érték: x szinusza.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
sin	+	+	+	+	+		+
sinl	+		+	+			+

sinh, sinhl — math.h

Szinusz hiperbolikusz értéket számol.

```
double sinh(double x);
long double sinhl(long double x);
```

Megjegyzés: A függvényt *bcd* és *complex* típusok is használhatják.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
sinh	+	+	+	+	+		+
sinhl	+		+	+			+

sprintf — stdio.h

Formattált kimenetet ír egy sztringbe.

```
int sprintf(char *buffer,
            const char *format[,argument, ...]);
```

Megjegyzés: Megegyezik a **print** függvénnyel, de a kimenetét a *buffer* memóriaterületen helyezi el.
Visszatérési érték: a kiírt bájtok száma a lezáró EOS karakter nélkül.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

sqrt, sqrtl — math.h

Négyzetgyököt számol.

```
double sqrt(double x);
long double sqrtl(long double x);
```

Visszatérési érték: az x négyzetgyöke.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
sqrt	+	+	+	+	+		+
sqrtl	+		+	+			+

srand — stdlib.h

Inicializálja a véletlenszám generátort.

```
void srand(unsigned seed);
```

Megjegyzés: a *seed* értékével a véletlenszám generátornak új kezdőértéket adhatunk.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

sscanf — stdio.h

Sztringből formattált adatot olvas.

```
int sscanf(const char *buffer,
           const char *format[,address, ...]);
```

Megjegyzés: Megegyezik az ***scanf*** függvénnyel, de a bemenetét a *buffer* által megadott memóriaterületről veszi.
Visszatérési érték: a sikeresen beolvasott és tárolt mezők száma.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strcat, _fstrcat — string.h

Sztringhez egy másik karakterláncot fűz.

```
char *strcat(char *dest, const char *src);
char far far*_fstrcat(char far *dest, const char far *src);
```

Megjegyzés: Az *src* sztringet hozzámásolja a *dest* végéhez.

Visszatérési érték: Az összefűzött sztringre mutató pointer.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
strcat	+	+	+	+	+	+	+
fstrcat	+		+				

strchr — string.h

Egy sztringben adott karakter első előfordulását keresi.

```
char *strchr(const char *s, int c);
```

Visszatérési érték: a *c* karakter *s*-beli első előfordulási helyére mutató pointer. Ha *c* nem található *s*-ben, NULL értékkel tér vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strcmp — string.h

Két sztringet hasonlít össze.

```
int strcmp(const char *s1, const char *s2);
```

Visszatérési érték: negatív, ha $s1 < s2$, nulla, ha $s1 == s2$ és pozitív, ha $s1 > s2$.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strcpy — string.h

Egy sztringet másikba másol.

```
char *strcpy(char *dest, const char *src);
```

Megjegyzés: Az *src* sztringet másolja a *dest* sztringbe a lezáró EOS karakterrel bezárólag.
Visszatérési érték: *dest* pointere.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

stricmp — string.h

Két sztringet hasonlít össze, kis- és nagybetűket azonosnak véve.

```
int stricmp(const char *s1, const char *s2);
```

Visszatérési érték: negatív, ha $s1 < s2$, nulla, ha $s1 == s2$ és pozitív, ha $s1 > s2$.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strlen — string.h

A sztring hosszát adja meg.

```
size_t strln(const char *s);
```

Visszatérési érték: az *s* sztring hossza a lezáró EOS karakter nélkül.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strlwr — string.h

Egy sztring nagybetűit kisbetűkre cseréli le.

```
char *strlwr(char *s);
```

Visszatérési érték: s.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strncat — string.h

Maximált hosszúságú karakterláncot fűz egy sztringhez.

```
char *strncat(char *dest, const char *src,
              size_t maxlen);
```

Megjegyzés: A *src* karakterláncból maximum *maxlen* karaktert fűz a *dest* sztring végéhez.
Visszatérési érték: *dest*.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strncmp — string.h

Összehasonlít két maximált hosszúságú karakterláncot.

```
int strncmp(const char *s1, const char *s2,
            size_t maxlen);
```

Megjegyzés: Megegyezik ***strcmp***-vel, de mindkét sztringből maximum *maxlen* karaktert vesz figyelembe.
Visszatérési érték: negatív, ha $s1 < s2$, nulla, ha $s1 == s2$ és pozitív, ha $s1 > s2$.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strncpy — string.h

Maximált hosszú sztring másolása.

```
char *strncpy(char *dest,const char *src,
              size_t maxlen);
```

Megjegyzés: A *src* sztringből maximum ***maxlen*** karaktert másol át *dest*-be.
Visszatérési érték: *dest*.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strset	**string.h**

Egy adott karakterrel feltölt egy sztringet.

```
char *strset(char *s, int c);
```

Visszatérési érték: s.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

strstr	**string.h**

Egy sztringben adott részlánc első előfordulását keresi.

```
char *strstr(const char *s1, const char *s2);
```

Visszatérési érték: az *s2* részlánc *s1*-beli első előfordulási helyére mutató pointer. Ha *s1* nem található *s2*-ben, NULL értékkel tér vissza.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

strupr	**string.h**

Egy sztring kisbetűit nagybetűkre cseréli le.

```
char *strupr(char *s);
```

Visszatérési érték: s.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+		+	+			+

system — stdlib.h

Az operációs rendszer egy parancsát hajtja végre.

```
int system(const char *command);
```

Megjegyzés: DOS alatt használva behívja COMMAND.COM fájlt, hogy végrehajtsa a *command* sztringben megadott DOS parancsot.
Visszatérési érték: nulla, ha sikeres a végrehajtás, különben -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+		+			+

tan, tanl — math.h

Tangens értéket számol.

```
double tan(double x);
long double tanl(long double x);
```

Megjegyzés: A függvény használható *bcd* és *complex* típusokkal is.
Visszatérési érték: tangens *x*.

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
tan	+	+	+	+	+	+	+
tanl	+		+	+			+

tanh, tanhl — math.h

Tangens hiperbolikusz értéket számol.

```
double tanh(double x);
long double tanhl(long double x);
```

Megjegyzés: A függvény használható *bcd* és *complex* típusokkal is

Visszatérési érték: tanh(x).

Portabilitás:

	DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
tanh	+	+	+	+	+	+	+
tanhl	+		+	+			+

tell — io.h

Lekérdezi az aktuális fájl-pozíciót.

```
long tell(int handle);
```

Visszatérési érték: az aktuális fájl-pozíció, hiba esetén -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

time — time.h

Az aktuális időt kérdezi le.

```
time_t time(time_t *timer);
```

Megjegyzés: az 1970. január elseje óta eltelt időt adja meg másodpercben **timer*-ben.

Visszatérési érték: az idő másodpercekben.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

toascii — ctype.h

ASCII karakterré alakít

```
int toascii(int c);
```

Megjegyzés: Az alsó 7 bit kivételével törli *c* összes bitjét.

Visszatérési érték: a c konvertált értéke.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

tolower ctype.h

Kisbetűtre alakít.

```
int tolower(int c);
```

Visszatérési érték: *c* konvertált értéke.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

toupper ctype.h

Nagybetűre alakít.

```
int toupper(int c);
```

Visszatérési érték: c konvertált értéke.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

ungetc stdio.h

Egy karaktert ír vissza az input folyamba.

```
int ungetc(int c, FILE *stream);
```

Visszatérési érték: A *c* karakter, hiba esetén EOF.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+	+	+	+

write — io.h

Alacsony szintű fájlírás.

```
int write(int handle, void *buf, unsigned len);
```

Megjegyzés: *len* bájtot ír a *handle*-vel azonosított fájlba a *buf* által megadott memóriából.
Visszatérési érték: A sikeresen kiírt bájtok száma, hiba esetén -1.

Portabilitás:

DOS	UNIX	Win16	Win32	ANSI C	ANSI C++	OS/2
+	+	+	+			+

F7. A lemezmelléklet használata

A lemezmellékleten a példaprogramok fejezetenként és témánként csoportosítva tömörített állományokban helyezkednek el. A TELEPIT.EXE program segítségével bonthatjuk szét azokat a tömörített állományokat, amelyekre szükségünk van. Az alábbiakban összefoglaljuk a példaprogramok telepítéséhez szükséges ismereteket.

A telepítés indításához aktuálissá kell tennünk azt a meghajtót, amelyik a lemezmellékletet tartalmazza, például az *A*:

```
A:
```

Ezután következhet a telepítő program indítása:

```
A:\>telepit
```

A telepítő program megjeleníti a lemezen található tömörített állományok listáját. A kifejteni kívánt állomány(ok) nevére rá kell állni a nyílbillentyűk (↑ és ↓) segítségével. A kijelölést, illetve a kijelölés megszüntetését a szóköz billentyű lenyomásával végezhetjük el (a kiválasztott állományok neve mellett a √ karakter látható). A válogatás során a jobb oldali ablakban rövid összefoglaló jelenik meg a kurzor melletti állomány tartalmáról.

A lemezmelléklet tömörített állományai két csoportba oszthatók:

- a P*n_nn_V3* nevű állományok az *n.nn* fejezet példaprogramjait tartalmazzák, amely a Borland C++ 3.1 rendszerében működnek,
- a P*n_nn_V5* nevű állományok az *n.nn* fejezet példaprogramjait tartalmazzák, amely a Borland C++ 5.02 rendszerében működnek.

Ha az állományok kijelölése után megnyomjuk az <*Enter*> billentyűt, akkor a az alábbi inputsor jelenik meg:

```
Hova kívánja a példákat telepíteni?
c:\cpp_prg
```

Az inputsorban meg kell adnunk annak a könyvtárnak a nevét, ahova a kijelölt fájlokat másolni szeretnénk.

Az alkönyvtár kiválasztása után (<*Enter*>) elkezdődik a példaprogramok kifejtése és másolása. A példaprogramok a tömörített fájl nevével azonos nevű alkönyvtárakba kerülnek.

Megjegyezzük, ha a telepítés előtt a lemezmelléklet tartalmát a merevlemez valamely alkönyvtárába másoljuk, majd onnan indítjuk a TELEPIT.EXE programot, akkor a telepítés sokkal gyorsabban végbemegy.

Megjegyezzük azt, hogy a Windows'95 alatt futó Borland C++ 5.02 szerkesztői ablakában a programban írt ékezetes betű eredményként nem jelenik meg. Ebből kifolyólag a program szöveges információit ékezet nélkül adtuk meg.

Irodalomjegyzék

Kondorosi Károly - László Zoltán - Szirmay-Kalos László:
Objektum-orintált szofverfejlesztés
ComputerBooks, Budapest, 1996

Kris Jamsa:
C++
Kossuth Kiadó, 1997

Borland C++: Programmer's Guide. Version 5.0
Borland International, 1996

Borland C++: Lnaguage Reference Vol. 1. Vol. 2. Version 5.0
Borland International, 1996

Brajrne Stroustrup:
The C++ programming Language
AT&T Bell Laboratories, Murray Hill, New Jersey

Benkő Tiborné - Benkő László - Tóth Bertalan:
Programozzunk C nyelven
ComputerBooks, Budapest, 1994

M.Waite - S.Prata - D.Martin:
C Primer Plus
Howard W. Sams and Co., Inc., 1984

Thomas Plum:
Tanuljuk meg a C nyelvet
Novotrade Rt., Budapest, 1987

B.W.Kernighan - D.M.Ritchie:
A C programozási nyelv
Műszaki Könyvkiadó, Budapest, 1988

Brian W.Kernighan - Dennis M.Ritchie:
The C programming language (second edition)
Prentice Hall, Inc., New Jersey, 1988

Clovis L. Tondo - Scott E. Gimpel:
C programozási gyakorlatok
Műszaki Könyvkiadó, Budapest, 1988

Mark Williams Company:
ANSI C. A Lexical Guide
Prentice Hall, New Jersey, 1988

Herbert Schild:
Using Turbo C
Osborne McGraw-Hill, 1988

Angie Hansen:
Learn C now
Microsoft Press, 1988

Benkő Tiborné - Urbán Zoltán:
IBM PC programozása Turbo C nyelven
BME MTI, Budapest, 1989

Kocsis Tamás - Poppe András:
C programozási feladatgyűjtemény és példatár
BME MTI, Budapest, 1992

Benkő László - Meskó László - Tóth Bertalan - Schuler László:
Programozás C és C++ nyelven: példatár
BME MTI, Budapest, 1992

N.Wirth:
Algoritmusok + Adatstruktúrák = Programok
Műszaki Könyvkiadó, Budapest, 1982.

Tárgymutató

#

#define, 114
#elif, 118
#else, 118
#endif, 118
#if, 118

*

*argv[], 111

_

_NOCURSOR, 243
_NORMALCURSOR, 243
_SOLIDCURSOR, 243

A

adatfolyam, 10
adjustfield, 37
alaptípusok
    double, 12
    float, 12
app, 105
arc, 247, 289
argc, 111
azonosítók, 11

B

bar, 247, 289
bar3d, 247, 289
bcd osztály, 273
bitmező, 77
blokk, 81
bool, 265

C

catch, 268
cerr, 33
char *env[], 111
cin, 10, 29
circle, 247, 290
class, 138, 140
cleardevice, 247, 290
clearviewport, 248, 290
close, 94
closegraph, 246, 291
clreol, 241, 279
clrscr, 241, 279
const_cast, 265
copy constructor, 149
cout, 10, 29
cprintf, 240, 241
cputs, 241

D

default, 58
delay, 286
delete, 129, 147, 159, 194
delline, 241, 279
destruktor, 149
detectgraph, 246, 291
dinamikus objektum, 158
do..while, 62
drawpoly, 247, 292
dynamic_cast, 266

E

early binding, 178
egységbezárás, 135
ellipse, 247, 292
előfeldolgozó, 114
else, 55
encapsulation, 135, 136
enum, 16, 109
explicit, 271

F

fail, 96
fájl változó, 93
false, 265
fill, 37
fillellipse, 247, 292
fillepoly, 293
fillpoly, 247
fixed, 33
floodfill, 247, 293
for ciklus, 59
formátumsztring, 26
free, 66
friend, 140, 165, 188, 193, 197
függvény, 81

G

generic class, 155
gerfillpattern, 250
get, 38, 93
getarccoord, 293
getarccoords, 247, 249
getaspectratio, 247, 249, 294
getbkcolor, 248, 249, 294
getche, 241
getcolor, 248, 250, 294
getdafaultpalette, 295
getdefaultpalette, 248
getdrivername, 250, 295
getfillpattern, 247, 295
getfillsettings, 247, 250, 296
getgraphmode, 246, 250, 296
getimage, 248, 297
getline, 102
getlinesettings, 250, 297
getmaxcolor, 248, 250, 298
getmaxmode, 250, 298
getmaxx, 250, 251, 298
getmaxy, 250, 251, 299
getmodename, 250, 299
getmoderange, 250, 299
getmodrange, 246
getpalette, 248, 250, 299
getpalettesise, 248
getpalettesize, 300
getpixel, 248, 300
gettext, 241, 279
gettextinfo, 242, 280
gettextsettings, 248, 250, 300
getviewsettings, 248, 301
getviewsettngs, 250
getx, 301
getxx, 250
getxy, 250
gety, 302
gotoxy, 241, 281
graphdefaults, 246, 302
grapherrormsg, 249, 302
graphfreemem, 246, 302
graphgetmem, 246, 303
graphresult, 249, 303

H

highvideo, 242, 281

I

if, 54
ifstream, 93
imagesize, 248, 303
inheritance, 135
initgraph, 239, 246, 304
inline, 143
insline, 241, 281
installuserdriver, 246, 306
installuserfont, 246, 306
int, 12

K

késői kötés, 178
kifejezések
  egyoperndusú, 40
  háromoperandusú, 40
  kétoperandusú, 40
komplex művelet, 273
komplex szám, 273
konstruktor, 148
konverziók
  aritmetikai, 50
  char, 51
  enum, 51
  int, 51
korai kötés, 178

L

late binding, 178
line, 247, 306
linerel, 247, 307
lineto, 247, 307
long, 109
lowvideo, 242, 281

M

main, 9, 67
malloc, 65
módosítók
    const, 22
    short, 12
    signed, 12
    unsigned, 12
    voletile, 22, 23
moverel, 247, 307
movetext, 241, 282
moveto, 247, 308
multiple inheritance, 165
mutable, 271
mutatók, 63

N

namespace, 269
new, 129, 147, 158
noreplace, 104
normvideo, 242, 282
nosound, 287

O

objektumpéldány, 144
ofstream, 93
open, 93
operator, 196
operátor
    postfix, 25
    prefix, 25
operator overloading, 192
operator overloading., 135
ostream osztály, 30
osztálysablon, 155
outtext, 248, 308
outtextxy, 248, 308

Ö

öröklés, 135, 165

P

pieslice, 247, 308
polymorphism, 135, 178
pow, 83
precision, 35
printf, 26
private, 139, 140, 145, 165, 200
project fájl, 233
protected, 139, 140, 146, 165
public, 139, 140, 165
put, 38, 93
putback, 38, 94
putimage, 248, 309
putpixel, 248, 309
puttext, 241, 282

R

read, 106
rectangle, 247, 310
referencia, 125
registerbgidriver, 310
registerbgifont, 310
reinterpret_cast, 266
restorecrtmode, 246, 310
return, 9, 81

S

scanf, 25
scientific, 34
sector, 247, 311
setactivepage, 247, 311
setallpalette, 248, 311
setaspectratio, 247, 312
setbkcolor, 248, 251, 313
setcolor, 248, 313
setcursortype, 283
setfillpattern, 247, 314
setfillstyle, 247, 314

setgraphbufsize, 246, 315
setgraphmode, 239, 246, 315
setlinestyle, 247, 315
setpalette, 248, 316
setrgbpalette, 317
settextjustify, 248, 317
settextstyle, 248, 317
setusercharsize, 248, 318
setviewport, 248, 319
setvisualpage, 248, 319
setviweport, 240
setwritemode, 319
showpoint, 34
showpos, 35
sizeof, 43, 54, 65, 72, 194
sokalakúság, 135, 178
sound, 286
sqrt, 83
static_cast, 267
stream, 10
struct, 72, 138
struktúra, 72
switch, 58
származtatott osztály, 165

T

template, 152
terminate, 269
textattr, 242, 283
textbackground, 242, 284
textcolor, 242, 284
textheight, 248, 320
textmode, 239, 242, 244, 285
textwidth, 248, 320
this, 144, 196, 202
throw, 268
típuskonverzió, 43
típussablon, 155
tömb, 71
true, 265
try, 268
type cast, 84
typedef, 24, 72, 139
typeid, 267
typeinfo, 267
typename, 267

U

union, 79
using, 270

V

vesszőoperátor, 48
virtual, 149, 178
virtuális tagfüggvény, 178
void, 10, 64, 82
volatile, 265

W

wchar_t, 270
wherex, 242, 285
wherey, 242, 285
while ciklus, 60
window, 240, 242, 286
write, 106

Z

zártság elve, 177

Megrendelőlap

Megrendelem az alábbi kiadványokat postai utánvéttel. Tudomásul veszem, hogy a postaköltség felszámításra kerül, és a szállítási idő 2-3 hét.

Utánnyomásoknál árváltozás lehetséges.

... *pld. Füzi János:* **3 dimenziós grafika és animáció IBM PC-n -** 1.480.-
... *pld. Füzi János:* **Interaktív grafika -** 1.993.-
... *pld. Jakab Zs.:* **Adobe PHOTOSHOP 4 magyar változat** 2.950.-
... *pld. Dr.Kovács T. -Dr.Kovácsné C.J -Ozsváth M.:* **Adatkezelés az MS ACCESS 2.0 alkalmazásával** 1.890.-
... *pld. Dr.Kovács T. -Dr.Kovácsné C.J.-Ozsváth M.:* **Adatkezelés az MS ACCESS 7.0 alkalmazásával W 95 alatt** 2.688.-
...*pld Nagy Gábor:* **Adattömörítési Kézikönyv** 1498.-
... *pld. Pintér M.:* **AutoCAD tankönyv** - DOS & WINDOWS; AutoCAD LT; AutoCAD R12 angol & magyar változathoz 899.-
... *pld. Pintér M.:* Rajzkészítés **AutoCAD R12-vel** 1.200.-
... *pld. Pintér M.:* Szilárdtestek modellezése **AutoCAD R12-vel** 1.200.-
... *pld. Pintér M.:* **AutoCAD Designer** 980.-
... *pld. Pintér M.:* **AutoVision** 1.961.-
... *pld. Benkő-Poppe-Benkő:* **Bevezetés a BORLAND C++ programozásába** 1.200.-
... *pld. Benkő L.-Benkő T.né-Tóth:* **Programozzunk C nyelven -** 1.499.-
...*pld. Benkő L. -Benkő T.-né:* **Programozási feladatok és algoritmusok TURBO C és C++ nyelven -** 1.998.-
...*pld Benkő T.né.- Poppe A.:* **Objektum-orientált programozás C++ nyelven-** 2.464.-
.. *pld. Dr.Kondorosi K.-Dr.László Z.-Dr.Szirmay-Kalos László:* **Objektum-orientált szoftverfejlesztés (C és C++)-** 2.616.-
... *pld. Dr.Dedinszky F.:* **CLIPPER 5 - 5.0, 5.01 és segédprogramjai** 1.200.-
... *pld. Nagy Z.-Spányik B.-Weisz T.:* **CorelDRAW! 5** 1.200.-
... *pld. Gazsó Z.:* **Adatbáziskezelés dBASE 5.0 for Windows rendszerben -** 1.989.-
... *pld. Gazsó Z.:* **Adatbáziskezelés FoxPRO-ban - 2.5, 2.6** verzió - Windows/DOS - 1.475.-
... *pld. Gazsó Z.:* **"VISUAL" Adatbázis-kezelők objektum-orientált programozása** - ***Visual dBASE; Visual FoxPro; Visual Object*** - 1.493.-
... *pld. Tamás-Kiss-Tóth:* **MS-DOS 6 - 6.2; 6.22 kiegészítéssel** 1.344.-
... *pld. Kovalcsik G.:* **EXCEL 5.0 for Windows kezdőknek * haladóknak** - magyar és angol változathoz 1.147.-
... *pld. Dr. Kovácsné C. J. - Ozsváth M.:* **EXCEL 5 függvényei** - magyar változathoz 990.-
... Kóczy J. **EXCEL 7 for Windows 95 felhasználóknak** 1994.-
... *pld. Krizsák László:* **EXCEL 7.0 programozása -** 1.456.-
... *pld Kovalcsik Géza:* **EXCEL 97 -** 2.490.-

A 175-35-91 telefonszámon tájékoztatjuk Önt a lakó- vagy munkahelyéhez legközelebbi szaküzletről, ahol kiadványainkat megvásárolhatja.

Ha a postai utat választja, kérjük a Megrendelőlapot levélcímünkre, **COMPUTERBOOKS Kft - 1253 Bp., Pf.: 71.** visszaküldeni.

Borland

Making Development Easier

JBuilder™

Borland® DataGateway for Java™

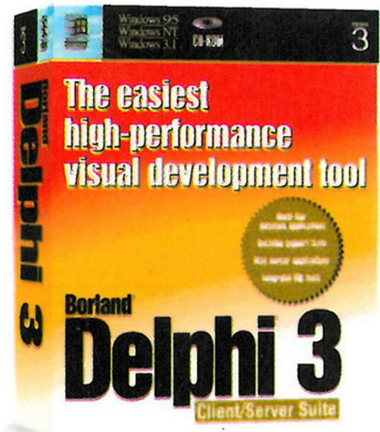
InterBase®

Visual dBASE®

Borland®/400

Delphi™3

C++Builder™



Borland® C++

IntraBuilder™1.5

SilVert

További információk web oldalunkon: www.borland.hu

Borland Magyarország Borland Magyarország, 1143 Budapest, Hungária krt. 79-81.
Telefon: 252-8145, fax: 252-8773, internet: http://www.borland.hu

ISBN 963-618-157-8

Ára: 2.464,- Ft (Áfával)