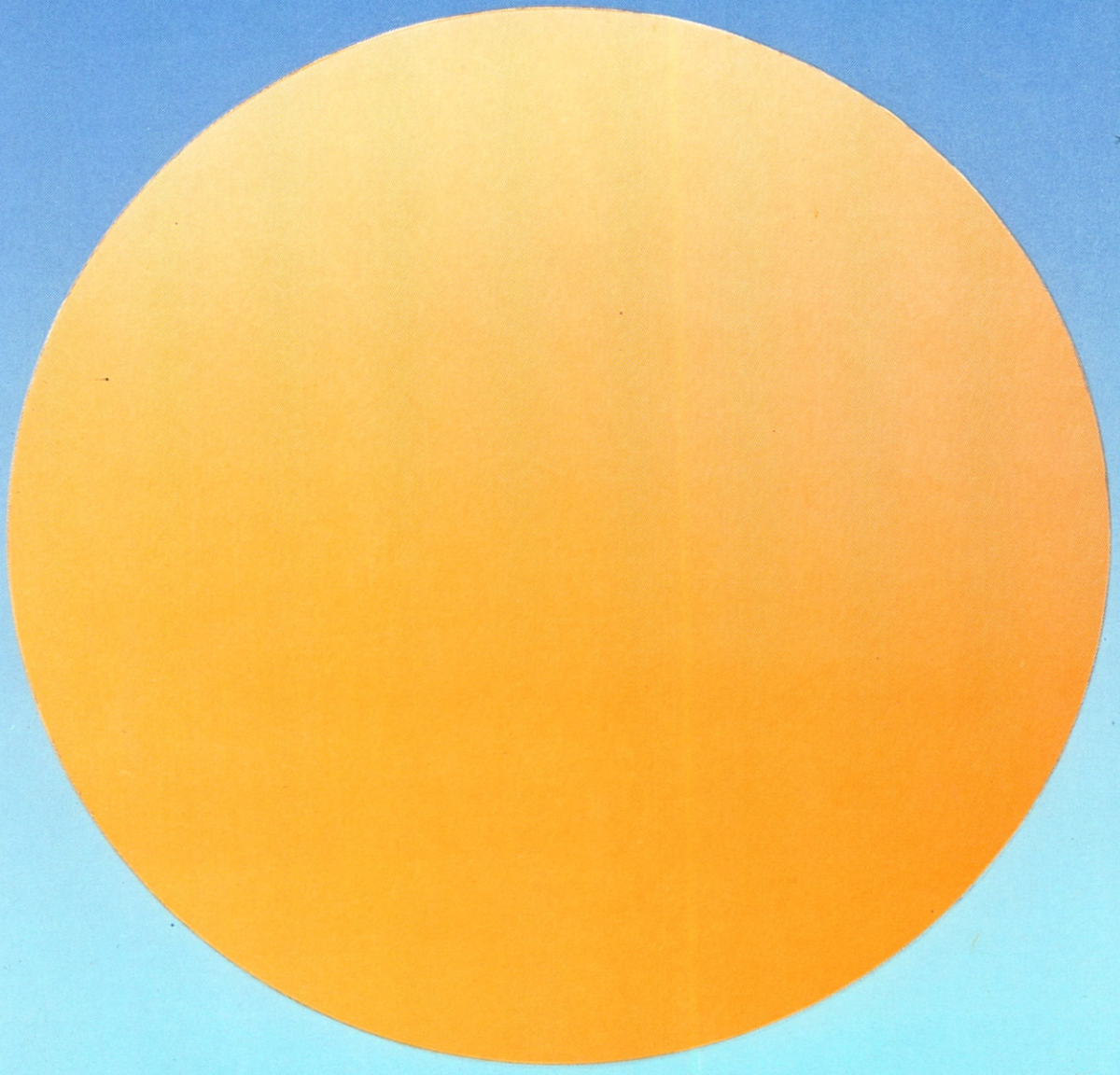
BENKŐ TIBORNÉ
POPPE ANDRÁS
BENKŐ LÁSZLÓ

BEVEZETÉS A BORLAND C++ PROGRAMOZÁSBA



COMPUTERBOOKS

BENKŐ TIBORNÉ
POPPE ANDRÁS
BENKŐ LÁSZLÓ

BEVEZETÉS A BORLAND C++ PROGRAMOZÁSBA

3. kiadás

LEKTOR
TÓTH BERTALAN

COMPUTERBOOKS
Budapest, 1995

A könyv a TEX szövegformázó rendszer felhasználásával COPAM típusú IBM PC kompatibilis számítógépen készült.

Kiadja: a *ComputerBooks* Kiadói Szolgáltató és Kereskedő Kft.
Felelős kiadó: a *ComputerBooks* Kft. ügyvezetője,
1126 Budapest, Tartsay Vilmos utca 12,
Telefon: 1751-564, 1753-591

ISBN 963 618 06 87

ÁFÉSZ Nyomda Vác, 165. 95

Tartalom

Előszó
Köszönetnyilvánítás

1 Bevezetés **1**
1.1 A BORLAND C++ fejlesztői környezet 3
1.1.1 Hardver és szoftver igények 3
1.1.2 Installálás 4
1.1.3 A BORLAND C++ fejlesztőrendszer kezelése 6
1.1.4 Szövegszerkesztés 12
1.1.5 Fordítás, programszerkesztés és futtatás 13
1.1.6 A project fogalma és használata 14
1.2 A menürendszer 16
1.2.1 Az ún. system menü 16
1.2.2 A **File** menü 16
1.2.3 Az **Edit** (szövegszerkesztés) menü 17
1.2.4 A **Search** (keresés) menü 18
1.2.5 A **Run** (futtatás) menü 18
1.2.6 A **Compile** (fordítás) menü 18
1.2.7 A **Debug** (nyomkövetés) menü 19
1.2.8 A **Project** menü 19
1.2.9 Az **Options** (rendszerjellemzők beállítása) menü . . 20
1.2.10 A **Window** menü 20
1.2.11 A **Help** menü 20

2 C programozás a BORLAND C++-ban **23**
2.1 Alapismeretek 24
2.2 A változók alaptípusai 26
2.2.1 A karakter típus és a sztringek 27
2.2.2 Az egész típusok 28
2.2.3 A felsorolt típus 29

2.2.4 Amit a logikai típusról tudni kell a C-ben 30
2.2.5 A lebegőpontos valós számok típusai 30
2.2.6 A mutató típusok 30
2.3 Az előfeldolgozó 33
2.3.1 Szimbólumok és makrók 33
2.3.2 Feltételes fordítás 37
2.3.3 Előredefiniált szimbólumok 38
2.3.4 File-beépítés 40
2.3.5 Implementáció-függő vezérlősorok 40
2.4 Tárolási osztályok, hatáskörök 41
2.4.1 Modulok, blokkok. 41
2.4.2 Függvények tárolási osztályai 42
2.4.3 Változók élettartama és hatásköre 42
2.4.4 Egyszerű adatdeklarátorok 44
2.4.5 Függvények definíciója és deklarációja 47
2.4.6 Módosító jelzők 50
2.4.7 Típusdefiniáló (`typedef`) azonosítók 52
2.5 Kifejezések 55
2.5.1 Elsődleges kifejezések 55
2.5.2 Operátorok 56
2.6 Konverziók 64
2.6.1 A konverzió a `char`, az `int` és az `enum` típusok között 64
2.6.2 Konverzió mutatók között 64
2.6.3 Aritmetikai konverziók 65
2.7 Utasítások 65
2.7.1 Kifejezés-utasítások 66
2.7.2 A feltételes utasítás 67
2.7.3 Ciklusszervező utasítások 69
2.7.4 Egyéb vezérlésátadó utasítások 71
2.8 Struktúrák és unionok 73
2.8.1 Struktúrák megadása 74
2.8.2 Hivatkozás struktúra elemekre 75
2.8.3 A bitmezők 76
2.8.4 A `union` fogalma 77
2.9 Mutatók és tömbök 79
2.9.1 A mutatók használata 79
2.9.2 Értékadás mutatóknak 83
2.9.3 Mutatók függvényparaméterként 84
2.9.4 Pointer aritmetika 85
2.9.5 Tömbök használata. Többdimenziós tömbök 87
2.9.6 Kapcsolat tömbök és mutatók között 88
2.9.7 Általános mutatók 91

2.9.8 Függvényekre mutató pointerek 92
2.10 Bevitel és kivitel 92
2.10.1 Alacsony szintű I/O 93
2.10.2 Folyam jellegű I/O 94
2.11 A `main` függvény 100
2.12 További magyarázatok 101
2.12.1 A balérték és jobbérték fogalma 101
2.12.2 Még egyszer a deklarációkról és a típusokról 102
2.12.3 Függvények változó számú paraméterrel 103
2.13 Összetett mintapélda 106
2.13.1 A tervezés egyes fázisai 106
2.13.2 A menükezelő rendszer listája 114

3 Programozás C++-ban **135**
3.1 Új elemek a C++-ban 136
3.1.1 Alternatívák a `#define` direktíva helyett 137
3.1.2 Cím szerint nyilvántartott típusú, vagy referencia típusú változók 138
3.1.3 Inicializált függvényparaméterek 141
3.1.4 C ++ kommentek 142
3.2 Az OOP alapjai 142
3.3 Egységbezárás 143
3.4 Öröklés 145
3.5 Többrétűség (polimorfizmus) 149
3.6 Függvénymezők definiálása 151
3.6.1 Függvénymezők aktivizálása 153
3.6.2 A `this` nevű, implicit mutató 154
3.7 Konstruktorok és destruktorok 155
3.7.1 Konstruktorok definiálása 156
3.7.2 Destruktorok definiálása 158
3.8 Mezőhozzáférés 159
3.8.1 Mezőhozzáférési szintek 160
3.8.2 Mezőhozzáférés és öröklés 161
3.9 Virtuális függvények 162
3.9.1 Késői összerendelés 162
3.9.2 Virtuális függvények deklarálása 166
3.10 Dinamikus objektumok 167
3.10.1 Dinamikus objektumok létrehozása 167
3.10.2 Dinamikus objektumok megszüntetése 168
3.11 További flexibilitás a C++-ban 169
3.11.1 Rokonok és barátok 169
3.11.2 Operator overloading 170

3.11.3 Példa egy operátor új jelentésének definiálására . . . 171
3.12 C++ I/O könyvtárak . . . 174
3.13 OOP megközelítésű rendszerfüggvények . . . 176
3.13.1 Komplex aritmetika . . . 176
3.13.2 BCD aritmetika . . . 177

4 IBM PC specifikus lehetőségek 181
4.1 Szöveg és grafika . . . 181
4.1.1 Programozás szöveges üzemmódban . . . 183
4.1.2 Programozás grafikus üzemmódban . . . 187
4.2 Overlay a BORLAND C++-ban . . . 196
4.2.1 Az overlay használata . . . 197
4.2.2 Swapping . . . 198

A Include file-ok és függvények 201
A.1 A BORLAND C++ include file-jai . . . 201
A.2 A könyvtári rutinok csoportosítása . . . 205
A.2.1 Karakter osztályozó rutinok . . . 205
A.2.2 Konverziós rutinok . . . 205
A.2.3 Katalóguskezelő rutinok . . . 205
A.2.4 Diagnosztikai rutinok . . . 206
A.2.5 Grafikai rutinok . . . 206
A.2.6 Input/Output rutinok . . . 207
A.2.7 Interface rutinok (DOS, 8086, BIOS) . . . 208
A.2.8 Memóriablokk- és sztringkezelő rutinok . . . 209
A.2.9 Matematikai rutinok . . . 210
A.2.10 Dinamikus tárkezelő rutinok . . . 211
A.2.11 Különleges rutinok . . . 211
A.2.12 Folyamatvezérlő rutinok . . . 212
A.2.13 Szabványos rutinok . . . 212
A.2.14 Karaktermódú képernyőkezelő rutinok . . . 212
A.2.15 Idő és dátum rutinok . . . 213
A.2.16 Változó argumentumlista kezelő rutinok . . . 213
A.3 Fontosabb könyvtári függvények . . . 214
A.3.1 Függvények szöveges üzemmódban . . . 214
A.3.2 Hangeffektusok létrehozása . . . 221
A.3.3 Függvények grafikus üzemmódban . . . 222
A.3.4 Általános könyvtári függvények . . . 252

Irodalomjegyzék 300

Előszó

A C programozási nyelv az egyik legnépszerűbb programfejlesztési eszköz a világon. A fejlesztők szívesen dolgoznak vele, mert általános célú, alkalmas igen nagy lélegzetű csoportmunkákban való felhasználásra (ellentétben a Pascal legtöbb implementációjával), nagyon jó hatásfokú kódot lehet vele előállítani, mégis magas szinten struktúrált, átfogóan szabványosított nyelv. Ez utóbbi azt jelenti, hogy egy adott géptípus adott operációs rendszerére kidolgozott – bizonyos programírási szabályokat figyelembe vevő – program viszonylag kis munkával, jól meghatározott helyeken való módosítással átírható más számítógép tetszőleges (a C nyelvet támogató) operációs rendszere alá. Itt azonban rögtön meg kell jegyeznünk azt is, hogy nagyon könnyű C nyelven áttekinthetetlen, nehezen megérthető és módosítható programokat készíteni. Nagyon fontos tehát a fegyelmezett, körültekintő programozási stílus alkalmazása, aminek az elsajátítása kb. annyi munkát igényelhet, mint maguknak a nyelvi elemeknek a megtanulása. Másképp megfogalmazva: a C nyelv nyújtotta szabadság csak annak válik igazán hasznára, aki megfelelően tud vele élni.

A C programozási nyelvet sokan tekintik struktúrált, gépfüggetlen assembly nyelvnek. Ez persze nem azt jelenti, hogy magasszintű nyelvről le kívánják "alacsonyítani", hanem éppen az előnyeit hangsúlyozzák: annyira gépközeli, hogy segítségével lehet akár operációs rendszert is írni, egyúttal annyira gépfüggetlen, hogy ugyanazt a rendszert más géptípusra forrásnyelvi szinten csekély módosítással át lehet tenni (kivéve persze a legalsó réteget képviselő hardver-közeli részeket, amelyek azonban úgyis assemblyben készülnek). A C nyelvet ez a képessége azonban nem korlátozza rendszerszintű programozásra, hanem egyúttal általános célú programfejlesztési eszközzé is kinőtte magát, amiben az említett nagyfokú szabványosság is komoly szerepet játszott.

Mint minden programozási nyelv, természetesen a C is fejlődik. E fejlődés eredményezte a C++ megszületését. Az AT&T-nél – a C nyelv születési helyén – már a 80-as évek elején komolyan foglalkoztak a C objektum-orientált programozási lehetőségekkel való kibővítésével. Akkor az "új" nyelvet

C extended with classes-nak nevezték, ebből született aztán a C++ megjelölés. A C++ fejlesztési koncepciója szerint az új nyelv majdnem teljes mértékben kompatibilis maradt a régivel. Ez biztosította, hogy megmaradjon a régi C gépközelisége, assembly nyelv jellege, ugyanakkor továbbfejlesztette a magasszintű nyelvi elemeket, elmenve egészen odáig, hogy a felhasználó által definiált tetszőleges absztrakt adatsruktúrákhoz automatikusan definiálódnak egyes műveletek, illetve lehetőség van arra, hogy a standard műveleteket az újabb adatstruktúrákra is értelmezzük.

Az objektum-orientált programozás (röviden OOP) lehetősége a C-t mostanra már egyértelműen a magasszintű programozási nyelvek családjának egyik leghatékonyabb tagjává tette. Gondoljuk csak meg: a C++-ban lehetőségünk van arra, hogy a BASIC tömbökhöz hasonló flexibilitású, futási időben dimenzionálható tömbtípust – például igazi sztringeket – definiáljunk, vagy a FORTRAN komplex számainak megfelelő adattípust hozzunk létre, és erre értelmezzük a szokásos operátorokat és a standard matematikai függvényeket.

Köszönetnyilvánítás

Könyvünk írásakor sok mindenben korábbi, hasonló jellegű munkáink szövegére támaszkodtunk. Éppen ezért itt újból meg szeretnénk köszönni Urbán Zoltánnak korábbi társszerzői, illetve lektori tevékenységét. A vele folytatott többszöri, kiterjedt megbeszélések nagyban hozzájárultak mondanivalónk letisztulásához, csiszolódásához.

Köszönet illeti Verhás Pétert is, aki `HION` nevű programját rendelkezésünkre bocsátotta. Ez a program tette lehetővé, hogy az IBM PC-n szerkesztett magyar nyelvű, ékezetes szövegfile-jainkat gond nélkül használhassuk a TEX szövegformázó rendszer LATEX makrócsomagjával. Sok segítséget nyújtott könyvünk elkészítéséhez Kiss Péter (KU Leuven) is: egyrészt építő javaslataival, másrészt a szöveg kinyomtatásával.

Nem hagyhatjuk szó nélkül családtagjaink megértő türelmét és segítőkészségét sem. Közülük sokan fáradtságot nem kímélve olvasták át a nyers kéziratot. Kérdéseikkel, észrevételeikkel jelentősen hozzájárultak könyvünk szövegének végleges formába öntéséhez.

Végül, de nem utolsó sorban meg szeretnénk köszönni Tóth Bertalan alapos lektori munkáját. Javaslatai, észrevételei igen hasznosak voltak számunkra. Különösen nagy segítséget jelentett munkája könyvünk ábráinak elkészítése során.

Budapest, 1991. Szeptember 25. *A szerzők*

1 Fejezet

Bevezetés

A C nyelvi rendszerek szinte kivétel nélkül *compiler*-ként kerülnek megvalósításra, azaz a forrásszövegből ún. tárgykódú modult (*object modul*) hoznak létre, amiből azután – egyéb tárgymodulok és könyvtárak (*libraries*) felhasználásával – az operációs rendszer szerkesztő programja (*linkage editor*) állítja elő a futtatható programot. Ennek előnye, hogy az egyes forrásfile-ok egymástól függetlenül fordíthatók, továbbá lehetőséget biztosít más nyelven (például assemblyben) megírt programrészek felhasználására is. Hátránya, hogy egy *módosítás–fordítás–kipróbálás–hibafelderítés* ciklus ideje általában hosszabb az értelmező (*interpreter*) típusú rendszerekénél. Megjegyezzük még, hogy vannak olyan C nyelvi fordítók is, amelyek tárgymodul helyett assembly forráskódot generálnak, illetve a legtöbb fordító rendelkezik ilyen opcióval. Mind a fordító, mind az adott assembly nyelv megismeréséhez hasznos gyakorlat ilyen fordítási listákat összevetni az eredeti forrásszöveggel.

A C nyelv előretörése természetesen nem "kímélte" a személyi számítógépeket sem. A nyelv sajátosságai miatt jó hatásfokú megvalósítása csak 16 illetve 32 bites gépeken lehetséges, ezért széles körben csak az Intel 8086 mikroprocesszorának megjelenése után beszélhetünk személyi számítógépes C fordítókról. Továbbiakban figyelmünket csak az IBM PC-vel kompatibilis számítógépekre fordítjuk. Erre a géptípusra is igen sok cég készített illetve készít C nyelvi fordítót. Ezek közül az MS-DOS (PC-DOS) operációs rendszer alatt a legnépszerűbbek: a Microsoft Corporation (MSC) és a Borland International (TURBO C illetve BORLAND C++) fordítói. Jelen könyv írásakor az általuk készített legfrissebb termékek az MSC 6.0 illetve a BORLAND C++ 2.0 verziók. Közös jellemzőjük, hogy rengeteg szolgáltatással és segédprogrammal rendelkeznek, kiterjedt függvénykönyvtáruk

van, és messzemenően figyelembe veszik az USA szabványügyi hivatalának, az ANSI-nak a legfrissebb szabványajánlatait.

A Borland cég termékei népszerűségüket magas fokú felhasználó-orientáltságuknak és sokszor meghökkentő sebességüknek köszönhetik, de nem mehetünk el szó nélkül a szokatlanul alacsony árak mellett sem. Emellett elsőként jelentkeztek a C nyelv olyan implementációjával, amely a *módosítás–fordítás–kipróbálás–hibafelderítés* ciklus valamennyi lépését egyetlen integrált fejlesztői rendszerben egyesítette. A BORLAND C++ rendszerben a Borland cég magát a fejlesztői környezetet is lényegesen továbbfejlesztette. Könyvünk lényegében ennek a rendszernek a példáján bemutatva az IBM PC-n történő C, illetve C++ programozás ismertetését tűzi ki célul. Az itt leírtakat azonban azok is haszonnal olvashatják, akik a BORLAND C++ rendszer hagyományos, parancsnyelvi változatát (`BCC`) kívánják használni.

A BORLAND C++ teljes implementációja az AT&T C++ fordítója 2.0-ás változatának, és megfelel az ANSI (American National Standards Institute) C-szabványának. A BORLAND C++ teljes mértékben kompatiblis Kernighan és Ritchie eredeti C definiciójával [1]. Ezen túlmenően a BORLAND C++ támogatja a kevert nyelvű és a különböző memória modellű programozást, illetve a Microsoft cég 3.0-ás verziójú Windows rendszerében futó programok fejlesztését is. Ez utóbbi lehetőség a BORLAND C++ rendszert igen hatékony eszközzé teszi a modern, interaktív-grafikus, ablakozós technikájú felhasználói felületek kialakításához. A BORLAND C++-ban írt alkalmazói programok az MS-Windows minden erőforrását elérhetik, így megfelelően hatékony hardver támogatás (i80286-os, de még inkább i80386-os, vagy i80486-os mikroprocesszorral rendelkező számítógépek) esetén igazi multitasking-ra, illetve virtuális memóriakezelésre mód nyílik. Maga a BORLAND C++ rendszer teljes mértékben ki tudja használni a fejlettebb mikroprocesszorok, illetve a nagy, több Mbyte-os memória nyújtotta lehetőségeket.

Könyvünkben a nyelv "klasszikus" részeinek ismertetésekor a *C* megjelölést fogjuk használni, *C++*-szal csak az objektum-orientált tulajdonságok ismertetésénél fogjuk illetni a nyelvet. A *BORLAND C++* kifejezést a Borland International cég által létrehozott nyelvi implementáció, illetve fejlesztői környezet megnevezésére használjuk. Bár könyvünk címe szerint a BORLAND C++ rendszerrel foglalkozik, igyekeztünk mondanivalónkat úgy leírni, hogy mindenki, aki általában a C, illetve a C++ nyelv megismerését tűzte ki célul, haszonnal forgathassa. Ez az egyik magyarázata annak, hogy komoly hangsúlyt fektettünk a portabilitási kérdésekre, és kissé háttérbe szorultak az IBM PC specifikus elemek. Ez alól kivételt jelentenek az MS-Windows alkalmazói programok fejlesztésével kapcsolatos ismeretek, melyekkel könyvünk második kötete foglakozik. Ezt a kivételezést az indo-

kolja, hogy az MS-Windows olyan modern, hardverkonfigurációtól független szoftver környezetet és felhasználói felület-szabványt teremt az IBM PC-n, amely filozófiáját és rendszerrutinjait tekintve nagyon közel áll a Magyarországon is egyre inkább elterjedő munkaállomásokon *workstation* (például a SUN, HP-Apollo, DEC gépeken) futó programok modern felhasználói felületeinek alapját képező X-Windows rendszerhez.

Kernighan és Ritchie 1978-ban írt eredeti C könyve felett – melynek magyar nyelvű változata [1] 1985-ben jelent meg (az 1988-as újabb magyar kiadás csak a 3 évvel korábbi szöveg változatlan utánnyomása) – eljárt az idő (no meg a szabványosítás). Éppen ezért 10 évvel az eredeti változat kiadása után a szerzőpáros elkészítette a könyv második, átdolgozott kiadását [2], amely már az 1983-as ANSI C szabvány szerint ismerteti a nyelvet. Sajnos ez a munka magyar nyelven publikusan nem hozzáférhető, így kénytelenek voltunk a szövegben alapvetően [1]-re hivatkozni.

A C++ nyelvre vonatkozó referenciaként ajánljuk Bjarne Stroustrup *The C++ programming language* című könyvét [3]. Ez a munka a C++-nak olyan "bibliája", mint Kernighan és Ritchie eredeti könyve volt a C esetében (bár a jelenlegi legfrissebb szabványok már mindkét könyv tartalmán túlmutatnak).

Az MS-Windows programok készítőinek ajánljuk a *Microsoft Windows Software Development Kit* dokumentációját [5], [6], és az IBM cég *Common User Access Advanced Interface Design Guide* című kiadványát [7].

1.1 A BORLAND C++ fejlesztői környezet

A következőkben bemutatjuk a BORLAND C++ fejlesztői környezetét. Az ismertetés során nem törekszünk teljességre, hiszen az egyes részletek a programrendszer eredeti dokumentációjának megfelelő köteteiben [8], [9] megtalálhatók.

1.1.1 Hardver és szoftver igények

A BORLAND C++ programrendszer használatához a következőkre van szükségünk: IBM PC kompatibilis számítógép min. 640 Kbyte RAM-mal, 80 karakter szélességű monitorral, merevlemezes egységgel (*hard disk*) és legalább egy hajlékonylemez meghajtóval (*floppy drive*). A lebegőpontos számítások sebességét nagy mértékben meggyorsítja egy 80x87 típusú aritmetikai társprocesszor (*arithmetic coprocessor*) használata. Az operációs

rendszer legyen a PC-DOS (MS-DOS) 2.0 vagy magasabb verziószámú változata. A teljes BORLAND C++ rendszer körülbelül 15 Mbyte helyet foglal el a merev lemezen. A BORLAND C++ támogatja a 640 Kbyte-on felüli memória (extended vagy expanded) használatát (lásd a függelékben).

A BORLAND C++ fejlesztői környezet egérrel is vezérelhető. használatát. Nem kötelező, hogy legyen egerünk, de ha van, akkor az alábbiak használatát javasolja a Borland cég:

- Microsoft Mouse 6.1 vagy későbbi verzió, vagy bármilyen, ezzel kompatibilis egér (például Genius Mouse),
- Logitech Mouse 3.4-es verzió vagy későbbi,
- a Mouse Systems cég PC Mouse 6.22-es vagy későbbi verziójú egere,
- IMSI Mouse 6.11-es verzió vagy későbbi.

1.1.2 Installálás

A programrendszert tartalmazó szoftvercsomagban a C++ fordító két változatát találhatjuk: az integrált fejlesztői környezetet tartalmazót és a parancssor orientált változatot. A BORLAND C++ rendszert a disztribúciós lemezeken található `INSTALL` program segítségével kell installálni, mert a programrendszer nagy része tömörítve található a disztribúciós lemezeken. Az `INSTALL` program biztosítja azt, hogy minden, a rendszer megfelelő használatához állomány rendelkezésre álljon, és hogy a programrendszer egyes paramétereinek alapértéke az adott hardverkonfigurációnak megfeleljen. További előnye az `INSTALL` program használatának az, hogy kialakítja a BORLAND C++ használatához szükséges optimális alkönyvtárrendszert a merevlemezen, az általunk megadott főkönyvtárban. Az `INSTALL` program egyes tevékenységei magától értetődőek (természetesen alapfokú angol nyelvtudás esetén), éppen ezért csak a legszükségesebb tudnivalókat írjuk le.

- Helyezzük az 1. számú installációs lemezt az `A:` jelű lemezegységbe.
- Adjuk ki a DOS-nak az `A:` meghajtóról az `INSTALL` parancsot.
- Az `INSTALL` program egy menüvel kezdeti installációs paramétereket kínál fel (például könyvtárnevek, példaprogramokat kicsomagolja-e, stb.) Ezen opciók egy része az *Enter* billentyű leütésével ki-, illetve bekapcsolható, más esetben egy-egy sztringet adhatunk meg (például könyvtárnevet).

Az installálást azzal fejezzük be, hogy az `AUTOEXEC.BAT` file-ban a `SET` paranccsal adjuk meg a `PATH` környezeti változóban a BORLAND C++

Borland C directory:	C:\BORLANDC
Include directories:	C:\BORLANDC\INCLUDE
Library directories:	C:\BORLANDC\LIB
Output directory:	C:\TMP

1.1. táblázat: A BORLAND C++ feltételezett könyvtárstruktúrája

keresési útvonalát. Tehát ha a C: merevlemez egységen a \BORLANDC főkönyvtárba installáltuk a C++ fordítót, akkor a

```
SET PATH=C:\BORLANDC\BIN
```

paranccsal állíthatjuk be a BORLAND C++ megfelelő DOS keresési útvonalát. Ekkor a BC paranccsal bármely könyvtárból elindíthatjuk a BORLAND C++ interaktív változatát. A BORLAND C++ rendszert mindig abból a könyvtárból célszerű indítani, amelyikben az adott témához kapcsolódó forrás-, illetve adatfile-okat kívánjuk gyűjteni. (Megjegyzendő, hogy plazma, vagy LCD képernyős laptop számítógépeken a rendszer indítása előtt célszerű kiadni a MODE BW80 DOS parancsot, vagy az INSTALL program lefutása után a BCINST segédprogram segítségével állítsuk be a fekete-fehér képernyő-üzemmód használatát.) A fenti főkönyvtár esetén, ha elfogadtuk az INSTALL program által felkínált értékeket, akkor a .h kiterjesztésű ún. include file-ok, vagy fejlécfile-ok (*include files, header files*) a C:\BORLANDC\INCLUDE alkönyvtárba, míg a .lib kiterjesztésű modulkönyvtárak és a gyári .obj kiterjesztésű segédmodulok a C:\BORLANDC\LIB alkönyvtárba kerülnek. A fordítás eredményeképpen keletkező .obj és a végső .exe programot a BORLAND C++ az ún. Output directory-ba helyezi. Ezt a paramétert vagy a mindenkori aktív könyvtárra (\.), vagy egy, az ideiglenes állományok tárolására szolgáló könyvtár nevére (például C:\TMP) érdemes beállítani. Ez utóbbi esetben megfontolandó, hogy a PATH-ba felvegyük-e a C:\TMP könyvtárt is, hiszen akkor a lefordított programjainkat az operációs rendszer prompt-jától is indíthatjuk. A továbbiakban az 1.1 táblázat szerinti struktúrát tételezzük fel.

Ha az installálás után mégis valamit változtattunk a könyvtárstruktúrán, tájékoztathatjuk a BORLAND C++ rendszert róla. A BC program elindítása után válasszuk ki az *Options* menüpont *Directories* almenüjét, és állítsuk be a könyvtári struktúránknak megfelelő neveket (lásd a fenti javaslatot).

Az INSTALL program sikeres befejeződése után feltétlenül olvassuk el a README file-t, amely a BORLAND C++-ra vonatkozó legfrissebb információkat tartalmazza. A BORLAND C++ főkönyvtárában megtaláljuk a

HELPME!.DOC file-t is. Ez a leggyakrabban felmerülő technikai jellegű kérdésekre ad választ. Természetesen olvassuk el a Borland cég ún. *No-Nonsence Licence Agreement*-jét is, és azzal egyetértve, azt aláírva juttassuk el a céghez a regisztrációs kártyát. Ezáltal a Borland cég bejegyzett felhasználói leszünk. Ez lehetővé teszi, hogy folyamatos tájékoztatást kapjunk a BORLAND C++ legújabb változatairól, a lehetséges verziócserékről (az ún. upgrade-ekről).

A rendszer installálása után javasoljuk, hogy még a gyakorlott TURBO C felhasználók is próbálják ki a BCTOUR programot, mely az 1.1-es táblázat szerinti könyvtárstruktúra mellett a C:\BORLANDC\TOUR alkönyvtárban található.

A BCTOUR végigvezeti a felhasználót a BORLAND C++ új integrált fejlesztői környezetén, és bemutatja a fejlesztői környezet új elemeit (ablakhasználat, egérkezelés, stb.).

Javasoljuk az Olvasónak, hogy könyvünket a BC programmal együtt használja; kipróbálva a szövegben található mintapéldáinkat.

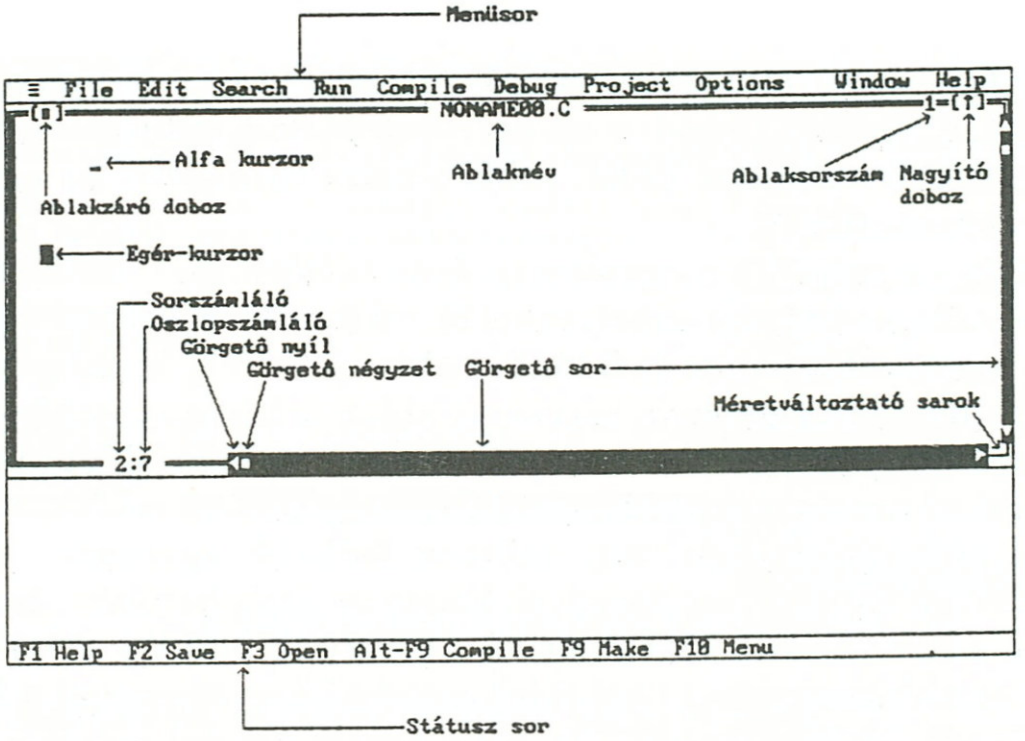
Megjegyezzük, hogy a BC, illetve a BCC programoknak létezik olyan változata, amely ki tudja használni az IBM PC 640 kbyte-on felüli memóriáját is. Ezen programváltozatok neve rendre BCX, illetve BCCX. A továbbiakban mi a BC, illetve BCC megjelölést fogjuk használni mindkét programváltozatra.

1.1.3 A BORLAND C++ fejlesztőrendszer kezelése

A BORLAND C++ integrált fejlesztői környezete (*Integrated Development Environment* – a továbbiakban *IDE*) ablaktechnikával (*window*) működik – az egyes programfunkciókhoz különböző ablakok tartoznak. Minden egyes ablak tetszőleges méretűre állítható és a képernyőn tetszőleges helyre áthelyezhető. Az integrált fejlesztőrendszer indítása a BC paranccsal történik. Az első indítás alkalmával a képernyő közepén egy ablakban megjelenik a BC program verziószáma, majd az *Enter* billentyű leütése után a képernyő az 1. ábrán láthatóhoz lesz hasonló.

A BC program mindig az utoljára szerkesztett forrásállományt az aktív editáló ablakba helyezve jelentkezik be. Természetesen az első használat alkalmával a BC még semmilyen forrásfile-t sem ismer, így egy NONAME00.C (00 sorszámú "nevenincs") file-t nyit meg. Az editáló ablak példáján keresztül mutatjuk be, hogy egy ablaknak milyen részei vannak, és hogy melyek az ablakkezelés alapjai az IDE-ben.

Minden ablaknak van egy neve, ez az *ablaknév*, ami az ablakkeret közepén, felül látható. Az IDE saját ablakain kívül (*Help window, Message window, Output window, Whatch window*, stb; lásd később) tetszőleges számú szerkesztő (editáló) ablakot nyithatunk. Minden egyes megnyitott



1.1. ábra: Egy ablak részei az editáló ablak példáján bemutatva

forrásállományhoz egy önálló ablakot rendel az IDE, ezen ablakok közül az első 9 darab egy-egy *ablaksorszámot* kap. Az editáló ablakok nevei a szerkesztés alatt álló file nevével egyeznek meg. Az ablakok további tartozékainak csak akkor van jelentősége, ha *egér* (*mouse*) is installálva van a számítógépünkön. (Az IDE kényelmes kezeléséhez az egér használatát javasoljuk.)

A képernyőn a jól ismert *alfanumerikus kurzor* mellett, ha egeret is installáltunk a számítógépünkhez, egy tömör téglalap is látható. Ezt *egér-kurzor*nak nevezzük. Az egér-kurzor az egér mozgatásával tetszőleges pozicióba helyezhető. A kényelmes ablakkezelés az egér-kurzor és az egér balodali nyomógombja segítségével lehetséges. A továbbiakban *kattintás*nak, vagy *klikkentés*nek hívjuk azt, ha az egér-kurzorral a képernyő adott pozicióján állva, egyszer, rövid ideig megnyomjuk az egér baloldali gombját.

Minden ablak bal felső sarkában szögletes zárójelek közt láthatunk egy kis tömör négyszöget [□], ez az *ablakzáró doboz* (*close box*). Az egérrel az ablakzáró dobozon kattintva az adott ablakot bezárhatjuk. Mivel ekkor az ablak tartalma elvész, szerkesztő ablakok zárásánál az IDE figyelmeztet minket. Minden ablak jobb felső sarkában egy szögletes zárójelek közt álló nyilacska látható ([↑]). Ez az ún. nagyító ikon, vagy *nagyító doboz* (*zoom box*). A [↑] alakú nagyító dobozon kattintva kattintva az aktuális ablak el fogja foglalni a képernyő teljes hasznosítható részét. Ha egy ablak a teljes, rendelkezésre álló képernyőterületet elfoglalja, akkor a nagyító dobozban egy kettős nyilacska található ([↕]). Ez az automatikus kicsinyítés lehetőségére utal. A nagyító dobozon kívül a *méretváltoztató sarok* – minden ablak

jobb alsó sarka – használható egy ablak "átszabására". Álljunk az egér-kurzorral erre a sarokra, majd a az egér baloldali gombját folyamatosan lenyomva tartva és az egeret le-fel, jobbra-balra mozgatva az aktív ablak mérete megváltoztatható.

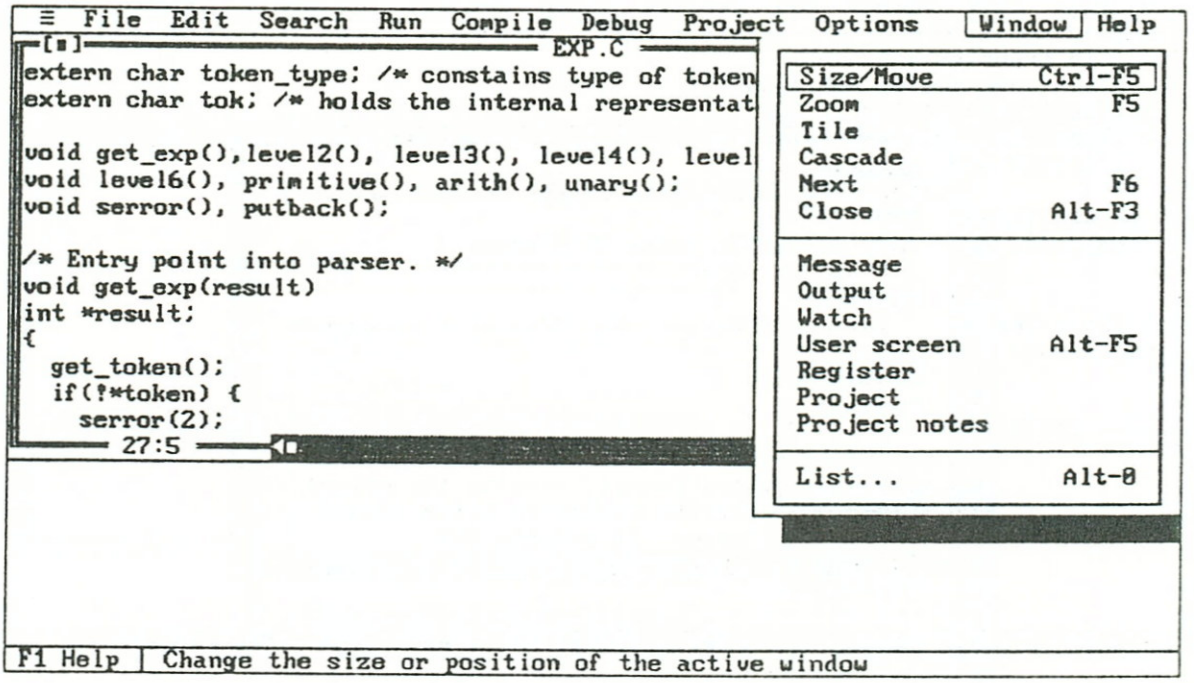
Az ablakok – a méretük megváltoztatásán túlmenően – természetesen a képernyő tetszőleges helyére áthelyezhetők. Álljunk az egérrel az áthelyezendő ablak keretének tetejére (például az ablak nevére), és az egér baloldali gombját folyamatosan lenyomva tartva az ablak a képernyőn követni fogja az egér mozgását. Egérrel nem rendelkező rendszereken az ablakkezelés a főmenü **W***indow* menüjének funkcióival lehetséges (lásd az 1.2. ábrát).

Az egyes ablakok részben vagy teljesen fedhetik egymást. Egyik ablakból a másikba az egér segítségével könnyen átléphetünk: kattintsunk az egérrel az aktivizálandó ablakban. A különböző editáló ablakok az *Alt* billentyű és az ablaksorszám (*1*-től *9*-ig) egyidejű lenyomásával a billentyűzetről is könnyen aktivizálhatók. Az *Alt-0* billentyűkombináció a **W***indow* főmenüpont **L***ist* funkcióját aktivizálja. Ekkor egy ún. dialógus dobozban (a dialógus doboz fogalmát lásd később) felsorolt létező ablakok közül a kurzorral választhatunk.

Ha egy ablakot keskenyre zsugorítottunk, akkor a forrásszövegnek (vagy a BC üzeneteinek, helpjeinek) csak egy kis része látszik egyszerre. Hogy mégis az ablak méretének megváltoztatása nélkül az aktuális szöveges állomány egyéb részeit megnézhessük, a szöveg *görgethető* (scrollozható). Egér nélküli rendszer esetén a kurzurmozgató nyilakkal (↓, ↑, ←, →), valamint a *PgDn*, *PgUp* billentyűkkel görgethetjük az aktív ablak tartalmát. Ha van egerünk, akkor a *görgető sorokban* (melyek az ablak alsó, illetve jobb szélső keretén találhatók) lévő *görgető nyilakkal*, illetve *görgető négyzetekkel* mozgathatjuk a szöveget (kattintással, illetve a baloldali gombot lenyomva tartva és az egeret mozgatva).

Az editáló ablakok alsó keretén további információ is található: az alfanumerikus kurzor aktuális poziciójára vonatkozó *sor* és *oszlop számláló*.

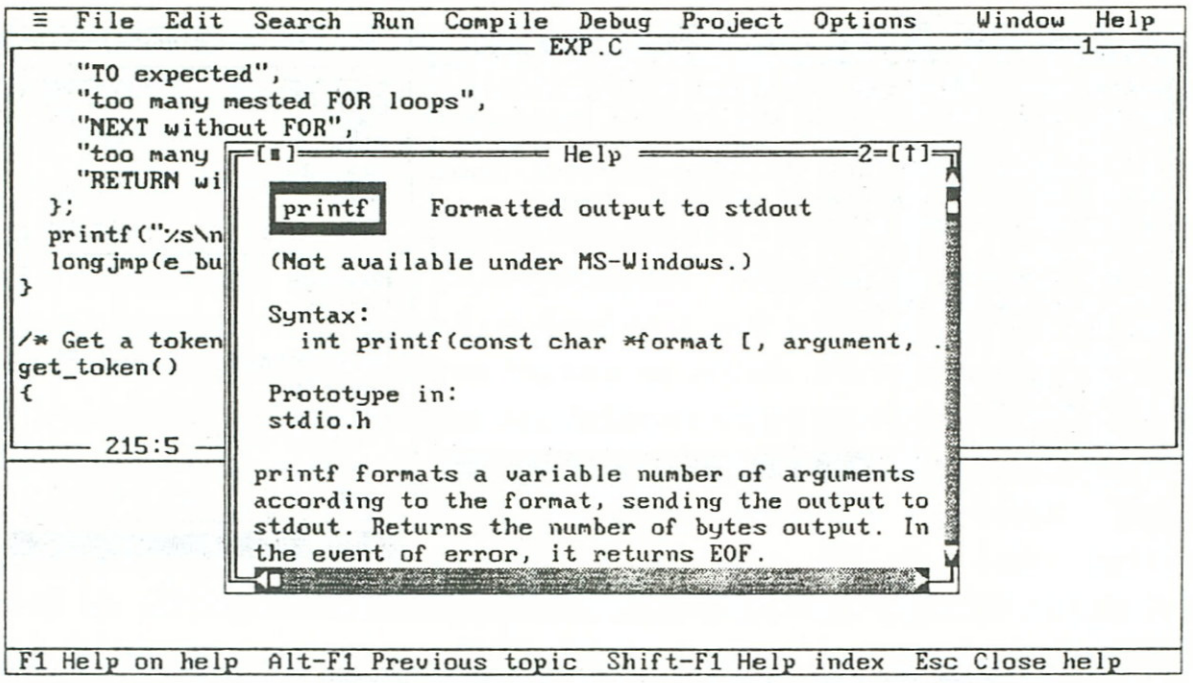
A képernyő tetején mindig látható a *főmenü* sora és a legalján az IDE aktuális állapotára vonatkozó *státusz sor*. A státusz sorban *kiemelve* látható billentyűk ún. *hot key*-k, azaz leütésük hatása azonnal jelentkezik. (A hot key elnevezés a jól ismert hot line = forró drót, azaz állandóan rendelkezésre álló telefonvonal kifejezés mintájára született.) Ilyen például az *F1*, melynek hatására a *Help* abľakban az aktuális szituációra vonatkozó help-szöveg jelenik meg (*context sensitive help*). Az *Esc* billentyű leütésével bármely művelet félbeszakítható. A főmenüben kiemelő színnel jelzett billentyűk az *Alt* billentyű egyidejű leütésével töltenek be *hot key* funkciót. A főmenü egyébként az IDE bármely helyéről az *F10* leütésével aktivizálható. Innen sorozatos kiválasztásokkal minden feladat hívható, mint például szövegszerkesztés, fordítás, különböző opciók beállítása, szimbolikus nyom-



1.2. ábra: A **W***indow* funkció pull down menüje

követés stb. A főmenü egyes pontjai a képernyő felső részén találhatók, aktiválásuk a kurzor rápozicionálásával és az *Enter* leütésével (vagy egy kattintással), vagy a kezdőbetű begépelésével történhet. (Ugyanez az elv érvényesül a további menük esetén is.) Egy főmenüpont kiválasztása után az adott funkcióhoz tartozó ún. *pull down* menü jelenik meg a képernyőn. Erre láthatunk példát az 1.2. ábrán: a főmenü **W***indow* menüjének egyes pontjai a különböző lehetséges ablakkezelő funkciókat jelentik.

Szövegszerkesztési állapotból – az alfanumerikus kurzor az egyik szerkesztő mezőben villog – a főmenübe az *F10* billentyű segítségével kerülhetünk. Bármely ponton lehetőségünk van a rendszer segítségét kérni az *F1* leütésével, a megjelenő tájékoztatás mindig pontosan arra a tevékenységre irányul, amit éppen kezdeményezünk vagy végrehajtunk. Ha tehát például szövegszerkesztés közben nyomjuk meg az *F1* billentyűt, akkor a szerkesztő parancsainak a listáját kapjuk. A help ablakban kiemelten megjelenő információkra rápozícionálhatunk a kurzor segítségével, kiválasztásuk az adott résztémához tartozó bővebb tájékoztatás megjelenítését eredményezi. Ha segítség közben újra *F1*-et ütünk, akkor a teljes tájékoztató hierarchia legtetejére, az ún. *help index*hez jutunk, és tetszőleges információt lekérdezhetünk sorozatos kiválasztásokkal. Nagyon hasznos segítőbillentyű a *Ctrl-F1*. Ha szövegszerkesztés közben lenyomjuk, akkor a kurzort éppen tartalmazó szóhoz (nyelvi alapelem, függvénynév stb.) kapcsolódó ismertetés jelenik meg. Az 1.3. ábra ezt a lehetőséget szemlélteti: szövegszerkesztés közben a `printf` függvényről kértünk felvilágosítást. A help-szövegben *kiemelve* megjelenő szavakkal kapcsolatban további információ kérhető: pozicionál-

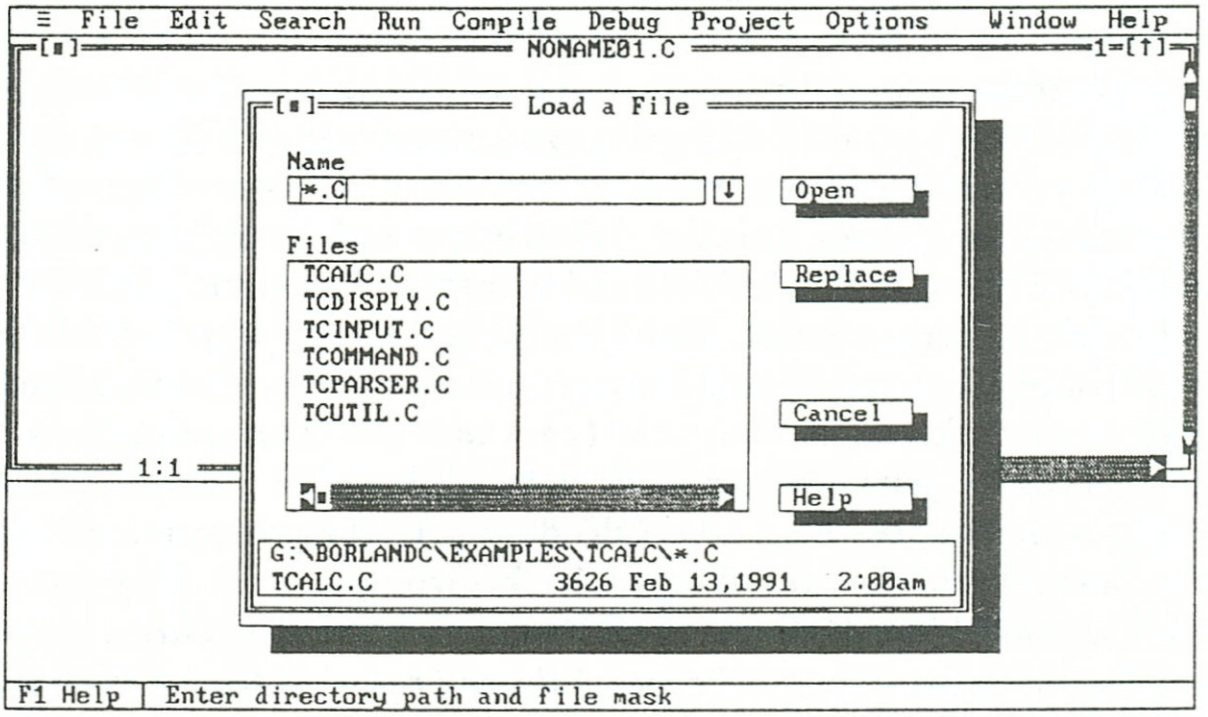


1.3. ábra: Help-ablak szövegszerkesztés közben.

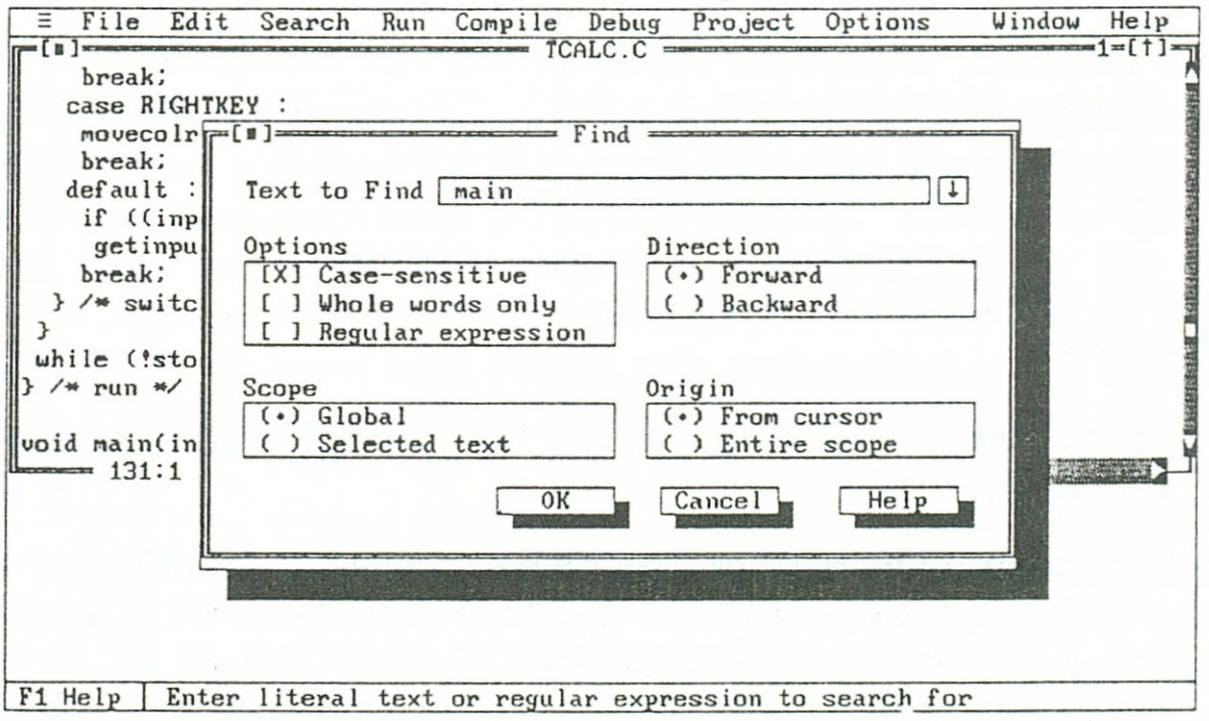
junk a kurzorral a megfelelő helyre és üssük le az *Enter*-t (vagy kattintsunk az egérrel). A C nyelvi elemekhez tartozó helpek nagy része mintapéldákat is tartalmaz, ezek szövegblokként kijelölve tetszőleges editáló ablakba másolhatók.

Az ablakok és menük mellett a BORLAND C++ fejlesztői környezet harmadik elemét az ún. *dialógus dobozok* alkotják. Erre láthatunk pédát az 1.4. és az 1.5. ábrán.

Egy dialógus doboz több különböző részből áll. A legfontosabb az ún. *input box*. Ez egy olyan kis, egysoros ablak a dialógus mezőben, ahol valamilyen információt közölhetünk az IDE-vel. Ilyen például egy megnyitandó forrásfile neve (`*.c` az 1.4. ábrán), vagy egy szövegben keresendő sztring (`main` az 1.5. ábrán). Ha az input box tartalmának valamilyen "előzménye" már van, akkor erre az input box melleti kis négyzetben lévő ↓ karakter utal. A korábban megadott sztringek a billentyűzet kurzorvezérlő gombjaival, vagy az egérrel "begörgethetők" az input boxba. Ha az input box tartalmával elégedettek vagyunk, ezt az *Enter* megnyomásával jelezhetjük. Az 1.4. ábrán az input boxban a file-neveket egy maszk segítségével adtuk meg. Ennek hatására az input box alatti lista mezőben az összes maszkra illeszkedő file-név megjelenik. A végső file-t a kurzormozgató gombokkal vagy az egérrel kell megjelölni és az *Enter* leütésével vagy egy kattintással választhatjuk ki. Minden dialógus dobozban található egy *OK*, *Cancel* és *Help* feliratú, árnyékolt mező. Ezek az un. *vezérlő gombok*. A kiemelt szinű betű leütésével vagy az egérrel a mezőn kattintva – mintha az illető vezérlő gombot lenyomtuk volna – az adott lehetőséget választhatjuk. *OK*



1.4. ábra: A főmenü **F***ile* | **O***pen* funkciójának dialógus mezője



1.5. ábra: A főmenü **E***dit* | **S***earch* funkciójának dialógus mezője

választásával azt közölhetjük az IDE-vel, hogy a dialógus mező aktuális tartalma szerinti paraméterekkel hajtsa végre az adott menüből kiválasztott műveletet (például egy file megnyitását, vagy egy sztring keresését). *Cancel* választása azt eredményezi, hogy az általunk eszközölt új paraméterbeállításokat a program "elfelejti", azaz olyan, mintha *Esc*-et nyomtunk volna. *Help* választásával segítséget kérhetünk a rendszertől az adott helyzetre vonatkozólag (olyan, mintha *F1*-et nyomtunk volna). Az 1.5. ábrán még két további paraméterbeállító dobozfajtát láthatunk. Az *Options* feliratú dobozban egyes opciók *Enter* leütésével (vagy egy kattintással) az un. *opcióbeállító dobozok* (*check boxes*) segítségével be- illetve kikapcsolhatók. Egy opció bekapcsolt állapotát (*checked option*) a szögletes zárójelek közötti x karakter jelzi: [x]. Ezeket a kis dobozokat vezérlő gomboknak (*check boxes*) nevezzük. Míg a vezérlő dobozokkal egyszerre több opciót is bekapcsolhatunk, addig vannak egymást kizáró opciók is. Az ilyen párosok külön-külön mezőkben láthatók (például az 1.5. ábrán *Direction*, *Origin*, *Scope*). Ezeket a mezőket *állító gomboknak* (*radio buttons*) nevezzük. Az éppen "élő" beállítást egy kerek zárójelekben lévő • karakter jelzi: (•). Az egyes opciókat beállító belső dobozok között a *Tab* billentyű leütésével tudunk közlekedni, az egyes opciók ki-/bekapcsolása a *SPACE* billentyű leütésével, vagy az adott helyen történő egér-kattintással lehetséges.

1.1.4 Szövegszerkesztés

A fenti ismeretek birtokában begépelhetjük első BORLAND C++ nyelvű programunkat. Ez – az egész világon elterjedten – a következő:

```
/* HELLO.C - elso kiserleti program */

#include <stdio.h>
void main()
{
    printf("Hello world!\n"); /* Udvozollek nagyvilag */
} /* end main() */
```

A fenti program beírásához lépjünk be a szövegszerkesztőbe a **F***ile* menü O*pen* pontjának kiválasztásával. Az ekkor jelentkező dialógus dobozban, az input boxban adjuk meg a `HELLO.C` nevet. *OK*-val fogadjuk el a dialógus doboz tartalmát. Ennek hatására egy új editáló ablak nyílik a `HELLO.C` file számára.

A BORLAND C++ szövegszerkesztője parancsaiban kompatibilis a régebbi TURBO C verziók szövegszerkesztőjével (Wordstar kompatibilis *Ctrl*-szekvenciák), de lehetőségünk van az egérorientált, menüvezérelt kezelésére is.

A szövegszerkesztő egy intelligens, ún. teljes képernyős szerkesztő (*full screen editor*), kezelését könnyű elsajátítani, és igen sok szolgáltatással rendelkezik. Példaképp említhetjük azt a hasznos parancsot, amellyel egymásba ágyazott zárójelek között lehet eligazodni, vagy a blokk-kijelölés lehetőségét, a sokrétű blokkszintű parancsokat illetve a kereső–helyettesítő parancsok rugalmasságát.

Figyeljük meg, hogy a szerkesztéshez nem áll rendelkezésünkre az egész képernyőfelület, hanem csak egy körülzárt része, az ún. szerkesztési ablak (*edit window*). E fölött az említett főmenü, alatta pedig igen sokszor a fordítási–programszerkesztési üzenetek ablaka (*Message window*) található (hacsak nincsen éppen lezárva).

Itt kell említést tennünk az ún. *clipboard*-ról. (A *clipboard* szó *hirdetőtáblaként*, vagy *üzenőtáblaként* fordítható.) Ez egy speciális, ideiglenes jellegű szerkesztő ablak, amely buffer funkciót tölt be. Az IDE azon ablakai, amelyekben szövegblokkokat jelölhetünk ki, a *clipboard* segítségével tudnak egymásnak szövegrészleteket átadni. Amikor egy szövegblokkot kijelölünk valahol, akkor azt az IDE a *clipboard* végére fűzi és a *clipboard*-on az lesz az aktuálisan kijelölt blokk. (A *clipboard* tartalma ugyanúgy szerkeszthető, mint egy "rendes" állomány.) Ezután egy szerkesztési ablakban a *Paste* (hozzáragasztás) parancsot (vagy a régi *Ctrl-K, C*-t) kiadva az illető blokk az alfanumerikus kurzortól kezdve az ablakba másolódik.

1.1.5 Fordítás, programszerkesztés és futtatás

Miután beírtuk fenti kis programunkat, *F10*-zel (vagy a *Ctrl-K*, utána pedig *D* billentyűsorozat leütésével) a főmenübe juthatunk, hogy lefordíthassuk és futtathatóvá szerkeszthessük példaprogramunkat. Miután ez igen gyakori páros, ezért erre a két lépésre egyetlen *hot key* szolgál, az *F9*. (Mint később látni fogjuk, ez a billentyű valójában a *project make* funkció közvetlen hívását végzi, de mivel nem adtunk meg projectnevet, ezért a rendszer az aktuális file-ból készít futtatható programot.) Ha a program beírásakor hibát követtünk el, a hiba oka és helye az üzenetek ablakában (*Message window*) olvasható. Ahogy mozgunk a kurzorral föl/le az üzeneteken, a hibát tartalmazó forrás-állomány szerkesztési ablakában automatikusan a hibás sor jelenik meg környezetével együtt. *Enter*-t ütve egy üzenetre, a szerkesztési ablakba kerülünk, és a kurzor éppen a hibára (vagy közvetlen környezetére) mutat! Ezt ennél a példánál talán nehéz igazán értékelni, de egy nagyobb – több tíz, vagy több száz soros – programnál felbecsülhetetlen segítséget jelent. Ha hibát kell javítanunk, utána ismét *F9*-cel kérhetjük a futtatható file előállítását.

A file-név kiterjesztésekkel kapcsolatban a következő konvenciót követi a BORLAND C++. A `.c` kiterjesztésű állományokat hagyományos C

programnak tekinti, és nem fogadja el a C++ nyelvi elemeket, hacsak az *Options | Compiler* menüben nem állítottuk be ennek az ellenkezőjét. Ha egy forrásállomány kiterjesztése `.cpp`, akkor azt automatikusan C++ programként kezeli a fordító (függetlenül a beállított fordítási opciótól). Fontos, hogy igyekezzünk ezt a konvenciót betartani, hiszen például a struktúrák más tulajdonságokkal bírnak a C++-ban, mint a C-ben. Így elképzelhető, hogy egy általunk hagyományos C programnak szánt, de `.cpp` kiterjesztésű állományban lévő forráskódból fordított és szerkesztett futtatható kód másképp fog viselkedni, mint amire számítunk.

Hibátlan fordítás esetén futtathatjuk a programot a *Run* (futtatás) menü *Run* parancsával, vagy az ennek megfelelő *Alt-F9* közvetlen billentyű-kombinációval. Egy pillanatra eltűnik a kép, majd a sikeres futás után újra az integrált rendszerben találjuk magunkat. Mivel a legtöbb program használja a képernyőt, ezért a BORLAND C++ rendszer egyidejűleg két képernyőt tárol: a sajátját és a felhasználói programét. A rendszer a program képernyőjére kapcsol minden futtatáskor, majd azt követően vissza a sajátjára. Ha ezután szeretnénk még látni a kiírt eredményeket (mint a mostani programunknál is), akkor az *Alt-F5 hot key*-vel kapcsolhatunk át a felhasználói képernyőre (*user screen*), és mindaddig az lesz látható, amíg egy tetszőleges billentyűt meg nem nyomunk. Így tekinthetjük meg programunk eredményét, ami jelen esetben egy barátságos köszöntés. Ha programunk csak alfanumerikus módban használja a képernyőt, akkor célszerű az ún. *Output window*-t megnyitnunk (a *Window* menü megfelelő funkciójával, lásd az 1.2. ábrát). Ekkor a programunk minden kiírt üzenete ebben az ablakban fog megjelenni. Ez igen hasznos lehetőség például a nyomkövetéssel történő hibakereséskor.

1.1.6 A project fogalma és használata

A fejezet hátralevő részében a project fogalmával és a BORLAND C++ által nyújtott ezirányú támogatással ismerkedünk meg.

A nagyobb programok több modulból, azaz több forrásfile-ból állnak, és csoportmunkánál ezeket esetleg különböző személyek is írják. Ezenkívül általában felhasználásra kerülnek korábban – vagy mások által – készített rutinok is lefordított formában, tárgykódú file-okként vagy könyvtárakba (`.lib` állományok) gyűjtve. Maga a végső program – BORLAND C++ szóhasználattal: a *project* – tehát nagyszámú file együttes figyelését követeli, ugyanakkor szeretnénk azt is elérni, hogy egy-egy módosítás után csak a szükséges fordításokat kelljen végrehajtani. Ennek eldöntése ugyanis nem feltétlenül triviális feladat, hiszen egy include file-ra – mint később látni fogjuk – több forrásfile is hivatkozhat, azaz egy `.h` állomány módosítása több `.c` forrásfile újbóli fordítását is szükségessé teheti. Előfordulhat az

is, hogy lemezrendezés vagy más gépre költözés esetén némely tárgykódú `.obj` file "elkallódik", ami linkelés során hibát okoz. Ezért ezt az esetet is figyelni kell, és ilyenkor is le kell fordítani a megfelelő forrásfile-okat.

A BORLAND C++ integrált fejlesztő rendszere a projectek kezelését csak akkor tudja támogatni, ha előzőleg közöljük vele azt, hogy az adott célprogram milyen összetevőkből épül fel, és milyen függőségi viszony van az egyes komponensek között. Ezt az információt project leíró file-ba kell tenni a *Project | Add item* funkcióval. (A BORLAND C++ `.prj` file-jai bináris file-ok, a korábbi TURBO C project file-okkal nem kompatibilisak. Mindez nem okoz túl nagy gondot, lévén a BORLAND C++ project file-ok nagyon könnyen és gyorsan előállíthatók, illetve a programrendszerben van olyan utility program – a `PRJCNVT` –, amely a korábbi project file-okat új az formára konvertálja.) Például ha a célprogram csupán a `prg1.c` és `prg2.c` forrásfile-okból áll, akkor a megfelelő `prg.prj` file-ba a *Project | Additem* funkció dialógus dobozában ezt a két file-t válasszuk ki a felkínált `.c` file-ok közül. Az így megadott információ a fejlesztőrendszer számára a következőket jelenti: az elkészítendő file (a project) neve `prg.exe`. Ehhez két tárgykódú állományt kell a linkernek átadni, ezek `prg1.obj` és `prg2.obj`. A `prg1.obj` a `prg1.c`, a `prg2.obj` pedig a `prg2.c` forrásfile lefordításával nyerhető. A *Compile* menü *Make exe* funkciója (*F9*) a végeredménytől visszafelé haladva ellenőrzi, hogy minden érintett állomány megfelelően friss-e. Tehát először feljegyzi az összes felépítő file (`prg1.c`, `prg2.c`, `prg1.obj`, `prg2.obj`) utolsó módosításának időpontját, és összeveti ezeket a `prg.exe` keletkezésének idejével. Ha ez utóbbi a legfrissebb, akkor nincs tennivalója, sem szerkesztés, sem fordítás nem szükséges. Ha ellenben valamely `.obj` file ennél későbbi, akkor új szerkesztés biztosan kell. Most az adott `.obj` file-t tekintve célnak, az őt létrehozó file-ok (jelen esetben a megfelelő `.c` forrásállomány) idejének vizsgálatával dönt arról, hogy elég-e a szerkesztés, vagy fordítás is szükséges. Hasonlóképpen, ha valamely `.c` file frissebb a `.exe` végeredménynél, akkor azt biztosan újra kell fordítani és szerkeszteni. Ezzel a stratégiával biztosítja a rendszer, hogy a szükséges műveletek – és csak azok – végre legyenek hajtva.

Elképzelhető, hogy a projectben egyes `.c` file-ok függenek más állományoktól (tipikusan `.h` include file-októl). Szemben a korábbi TURBO C-vel, a BORLAND C++ az első *Make* alkalmával felméri az adott project függőségi viszonyait és ezt az információt automatikusan elhelyezi a project file-ban. A projectre vonatkozó fordítási paraméter-beállításokat (például a memóriamodellt) is a projectfile-ban tartja nyilván a rendszer.

Néha előfordulhat, hogy a file-ok dátumától függetlenül szükséges minden forrásfile-t újra lefordítani, például ha memóriamodellt váltottunk, vagy egyéb fordítási, programszerkesztési opciót megváltoztattunk. Ilyenkor a *Compile* menü *Build all* funkcióját kiválasztva, a projectleíró alapján,

de a dátumoktól függetlenül minden forrásfile-t újrafordít a rendszer, majd összeszerkeszti a célprogramot.

1.2 A menürendszer

A következőkben röviden ismertetjük a főmenü egyes pontjaival elérhető lehetőségeket.

1.2.1 Az ún. system menü

A főmenünek ez az első pontja, amelynek három állandó, előredefiniált funkciója van. Ezek az **A***bout* – egy dialógus dobozban közli a program verziószámát, a C*lear desktop* – törli az IDE munkafelületét, az ún. *desktop*-ot, azaz lezárja az összes ablakot, törli a *clipboard*-ot és a dialógus dobozok listáit és a project állapotára vonatkozó információkat –, valamint a **R***epaint* – újragenerálja az IDE képernyőjét. Ez utóbbi funkció például programok nyomkövetésénél lehet nagyon hasznos.

A system menü második felét mi magunk definiálhatjuk az **O***ptions* menü **T***ransfer* funkciójával (a részleteket lásd ott). A dolog lényege az, hogy más, kiegészítő programokat is meghívhatunk a BORLAND C++ integrált fejlesztői környezetéből anélkül, hogy kilépnénk az IDE-ből, ezáltal a segédprogramok úgy tűnnek, mintha maguk is az IDE részei lennének. Tipikus segédprogramok a `GREP` (sztring-minta keresése file-okban), a `TD` (TURBO Debugger - hibakereső program), a `TASM` (TURBO Assembler) stb. (E segédprogramok mind megtalálhatók a BORLAND C++ disztribúciós lemezeken, installálásuk opcionális az `INSTALL` programban.)

1.2.2 A F*ile* menü

Általános file-kezelési és operációs rendszerrel összefüggő műveletek tartalmaz. Az **O***pen* opcióval (*F3*-as *hot key*) egy lemezen tárolt forrásnyelvi file-t nyithatunk meg.

A kiválasztás során megadunk egy file-név maszkot (alapértelmezésben `*.cpp`), majd a rendszer felkínálja az összes, az adott maszknak megfelelő file-t, amelyek közül a kurzorral és az *Enter* billentyűvel választhatunk. A maszk utalhat más meghajtóra és/vagy más alkönyvtárra is, de a rendszer a maszkra illeszkedő file-okon kívül felkínálja a létező könyvtárakat is, és azok kiválasztásával is mozoghatunk a könyvtár-struktúrában fel és le egyaránt. Ha a maszk nem tartalmaz ? vagy * karaktert (egyértelmű névmegadás), akkor természetesen nincs szükség választásra.

A *New* menüpont egy új szerkesztési ablakot nyit `NONAMEXX.CPP` névvel, ahol `XX` a *New*-val létrehozott, egyelőre még névtelen forrásállomány sorszáma. Ez egy új, üres file editálását teszi lehetővé. (Természetesen ha C programot szeretnénk szerkszteni, akkor a maszkot az alapértelmezés szerinti `*.cpp`-ről át kell írnunk `*.c`-re.)

A *Save* menüpont (*F2*-es *hot key*) segítségével lemezre írhatjuk az aktuálisan szerkesztett állományt. Itt jegyezzük meg, hogy minden olyan esetben, amikor a szerkesztési ablakot valamilyen oknál fogva bezárjuk és az aktuális szöveg a legutolsó mentés óta módosítva lett, a rendszer automatikusan felkínálja a mentést. Ugyanezt teszi a *File* menü utolsó pontjánál, a *Quit*-nél is, amelyikkel elhagyhatjuk a BORLAND C++ rendszert.

A *Save as ...* menüpont is erre szolgál, de lehetővé teszi, hogy új nevet adjunk a mentéshez.

A *Change dir* segítségével új aktuális alkönyvtárra válthatunk, a *Print*-tel az aktív szerkesztési ablak tartalmát nyomtathatjuk ki a sornyomtatón, míg a *DOS shell* menüpont lehetőve teszi, hogy ideiglenesen elhagyjuk az integrált fejlesztői környezetet, és az operációs rendszernek adjunk parancsokat. A *Get info* menüpont kiválasztásával megtudhatjuk, hogy az integrált BORLAND C++ fejlesztő rendszer a számítógépünk erőforrásaiból mit és mennyit vesz igénybe.

A *Quit* menüpont segítségével hagyhatjuk el véglegesen a BORLAND C++ fejlesztői rendszert. E menüpont kiválasztása ekvivalens az *Alt-X hot key* használatával. Kilépéskor az *Options* | *Environment* | *Autosave* pontban beállítottak alapján automatikusan elmenti a rendszer a fejlesztői környezet állapotát (a szerkesztett forrásállományokat, a *clipboard*-ot, a *project file*-t, a nyitott ablakok listáját, stb.).

1.2.3 Az *Edit* (szövegszerkesztés) menü

Ez a menü kiegészítő szerkesztési műveletekre nyújt lehetőséget. A *Restore line* menüpont segítségével az éppen szerkesztett állományban az utoljára módosított sor tartalma állítható helyre.

A *Cut* menüpont kiválasztásával (*Shift-Del*) az aktív editáló ablakban kijelölt szövegblokkot kivághatjuk a kérdéses szövegből. A *Cut*-tal kivágott szövegeket a rendszer mindig a *clipboard* végéhez hozzáfűzi, és ez a blokk lesz a *clipboard* aktuális kijelölt szövegblokkja.

A *Copy* menüpont (*Ctrl-Ins*) az aktív editáló ablak kijelölt szövegblokkját érintetlenül hagyva elhelyezi azt a *clipboard*-on. A *clipboard* kijelölt blokkját a *Paste* (*Shift-Ins*) funkcióval az aktív editáló ablak alfanumerikus kurzorához másolhatjuk. A *Copy example* funkcióval egy C nyelvi elem helpjének blokként kijelölt mintaprogramját helyezhetjük el a *clipboard*-on.

A **Show** *clipboard* funkcióval a *Clipboard* ablakot aktivizálhatjuk, és így megnézhetjük az eddigi blokkmásolásaink "történetét".

1.2.4 A *Search* (keresés) menü

A korábbi TURBO editorok keresési funkciói kaptak helyet ebben a menüben. Bizonyos menüpontok szövegszerkesztés közben a régi *Ctrl* szekvenciákkal (például *Ctrl-Q, A*) is hívhatók, de a kapcsolatot a felhasználóval már az 1.5. ábrán is bemutatott dialógus dobozokon keresztül tartják. A **Search** menü pontjai a következők: **Find**, **Replace**, **Search again**, **Go to line number**, **Previous error** (*Alt-F7*), **Next error** (*Alt-F8*) és **Locate function**. Az utóbbi három a programfordítással (fordítási hibaüzenetek keresése a *Message window*-ban), illetve a **Debug** funkcióval kapcsolatos.

1.2.5 A *Run* (futtatás) menü

Ha kiválasztjuk ezt a főmenüpontot, akkor láthatjuk, hogy minden utasításahoz tartozik közvetlen billentyű-kombináció – lévén ez egy igen fontos menüpont –, és némi gyakorlás után már szinte kizárólag ezeket fogjuk használni. A **Run** (*Ctrl-F9*) parancs futtatja az aktuális programot, de ha az a legutolsó fordítás óta módosítva lett, akkor automatikusan megtörténik a programfordítás–szerkesztés is. A **Program reset** (*Ctrl-F2*) opció inicializálja a programunkat, erre akkor lehet szükség, amikor a végrehajtást felfüggesztettük nyomkövetés céljából, és újra elölről szeretnénk indulni. A **Go to cursor** (*F4*) segítségével nyomkövetési állapotba kapcsolunk – ha még nem voltunk ott, és egyidejűleg töréspontot helyezünk el a kurzor által mutatott utasításra, majd futtatjuk programunkat. A **Trace into** (*F7*) soronként hajt végre egy-egy utasítást oly módon, hogy végigléptet a meghívott függvényeken is, míg a **Step over** (*F8*) a függvényhívásokat egy egységként kezelve lépi át. Az **Arguments** menüponthoz tartozó dialógus dobozban a futtatandó programunk parancssor paramétereit állíthatjuk be (mintha a programot a DOS-ból hívnánk meg, egyedül a kimenet/bemenet átirányítást nem alkalmazhatjuk).

1.2.6 A *Compile* (fordítás) menü

Ez a főmenüpont tartalmazza a fordítással és szerkesztéssel kapcsolatos parancsokat. A **Compile to obj** utasítás segítségével kérhetjük az aktuális forrásállomány lefordítását szerkesztés nélkül, a **Link exe file** pedig a programszerkesztőt hívja. A **Remove messages** törli a fordítási-linkelési üzeneteket tartalmazó ablakot (a *Message window*-t). A többi parancs használatát külön, az 1.2.8-as pontban, a projectekkel kapcsolatosan ismertetjük.

1.2.7 A *Debug* (nyomkövetés) menü

A BORLAND C++ integrált fejlesztői környezet szerves része egy intelligens, szimbolikus, forrásnyelvi nyomkövető (*debugger*) is. Kezelése nem korlátozódik kizárólag erre a menüpontra, mert például a **Run** menünél is láttunk már nyomkövető utasításokat. A debugger használatához feltételenül szükséges, hogy a programjainkat az *Options | Compiler| Code generation | OBJ debug information* opció bekapcsolt (*On*) állása mellett fordítsuk, ellenkező esetben a nyomkövetéshez szükséges információk nem kerülnek bele a `.obj` illetve `.exe` file-okba.

Az *Inspect* (*Alt-F4*) funkció lehetővé teszi, hogy tetszőleges típusú változók tartalmát, vagy függvényparaméterek aktuális értékeit megtekinthessünk. Az *Evaluate* (*Ctrl-F4*) menüpont segítségével nyomkövetés közben tetszőleges C nyelvi kifejezés értékét lekérdezhetjük, illetve ha az balérték (lásd 2.12.1-nél), akkor módosíthatjuk is. A *Call stack* (*Ctrl-F3*) megadja azt a hívási sorozatot, amellyel az aktuális függvényig eljutott a programunk. A *Toggle breakpoint* (*Ctrl-F8*) és a *Breakpoints* menüpontok a töréspontok kezelésére szolgálnak: elhelyezhetünk, illetve megszüntethetünk töréspontokat adott forrásnyelvi soroknál, valamint törölhetjük, vagy egyenként lekérdezhetjük az összes meglévő töréspontot. A *Watch* menüpont egy újabb almenüt takar. A *Watch* almenübe foglalt szolgáltatások is a beépített nyomkövető kezeléséhez tartoznak: segítségükkel az ún. megfigyelőpontokat kezelhetjük (sorrendben újat vehetünk fel, törölhetünk, illetve módosíthatunk egy meglévőt és megszüntethetjük az összeset). A megfigyelőpontok az *Evaluate* parancshoz hasonlóan tetszőleges C nyelvi kifejezéseket tartalmazhatnak, de ezek értékét a nyomkövető minden lépés után automatikusan frissíti.

1.2.8 A *Project* menü

A projectek fogalmát külön fogjuk tárgyalni, itt csak a kezelésükre szorítkozunk. Az *Open project* opcióval jelölhetünk ki egy `.prj` project file-t, a *Close project* pedig törli az aktuálisat. Az *Add item* menüpont segítségével vehetünk fel egy újabb modult a project listába (project file-ba), a *Delete item* egy adott modult töröl a projectből. A *Local options* segítségével az *Options* menüben beállított fordítási opciókat bírálhatjuk felül egy adott programmodulra vonatkozólag. Az *Include files* menüpont segítségével nyerhetünk információt arról, hogy melyik forrásmodul milyen file-okat épít be, feltéve persze, ha a *Compile* menüből sikeresen végrehajtottuk a *Build all* vagy a *Make EXE* funkciót.

A projectet és a hozzá kapcsolódó ismereteket az 1.1.6-os rész tárgyalja részletesen.

1.2.9 Az *Options* (rendszerjellemzők beállítása) menü

Kiterjedt almenürendszerrel rendelkező menüpont. A *Full menus* pontban választhatunk, hogy teljes (*On*) vagy redukált menükkel (*Off*) dolgozzon az IDE. A *Compiler* almenü a fordítással kapcsolatos opciók beállítását teszi lehetővé, a *Linker* almenü pedig a szerkesztési opciókét. Melegen ajánljuk, hogy a *Compiler* almenü *Errors* almenüjében kapcsoljuk be az összes figyelmeztető üzenet kiíratását (ezek közül néhány alapértelmezésben *Off*-ra van állítva). Meglátjuk, hogy segítségükkel már fordítási időben kiszűrhetjük hibáink nagy részét!

A *Transfer* menüpont segítségével definiálhatunk új elemeket a system menü számára. Itt adhatjuk meg, hogy a system menübe újonnan felvett segédprogramoknak milyen hívási környezetre van szüksége, az IDE milyen dialógus dobozokat használjon, milyen névvel és rövidítéssel szerepeljenek a menülistában stb.

Az *Environment* almenü a szövegszerkesztéssel kapcsolatos opciók beállítására szolgál. A *Directories* segítségével módosíthatjuk az installáció során, vagy később a `BCINST` programmal beállított könyvtári struktúrát. És végül a *Save* menüpont a beállított opciók lemezre mentését végzi.

1.2.10 A *Window* menü

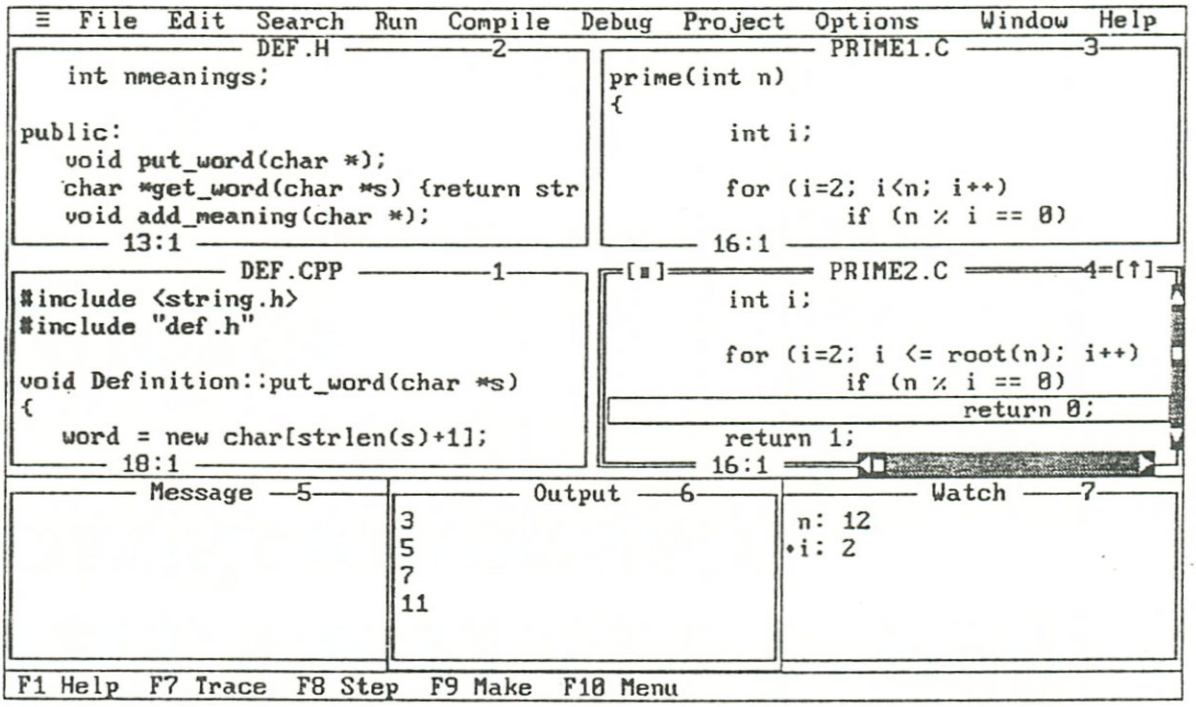
Ennek a menünek a legtöbb funkciójáról már szóltunk az 1.1.3-as részben. Most azokról a menüpontokról ejtünk néhány szót, amelyek kimaradtak az eddigi ismertetésből.

A *Tile* funkcióval elérhetjük, hogy az IDE az összes létező ablakot átfedés nélkül helyezze el a képernyőn. Ezt illusztrálja az 1.6. ábra. Az ilyen ablakelrendezés nagyon hasznos lehet egy program nyomkövetésénél: egyszerre láthatjuk például a forrásállományt, a program kimenetét és az egyes változók aktuális értékét. A *Cascade* funkció éppen ellentétes módon működik: a meglévő ablakokat egy kiterített kártyacsomaghoz hasonlóan, egymást átfedve helyezi el a képernyőn. A *Next* menüponttal a "következő" ablakba léphetünk be.

A *Message*, *Output*, *Watch* és *Register* opciókkal rendre az üzenetek, a program kimenet, a megfigyelőpontok és a CPU regiszterek ablakát nyithatjuk meg.

1.2.11 A *Help* menü

Ebben a menüben 5 funkció található. *Contents* segítségével kaphatjuk meg a help-rendszer tartalomjegyzékét, *Index* (*Shift-F1*) választásával a



1.6. ábra: A **W***indow* | **T***ile* funkcióval elrendezett ablakok

help-rendszer kulcsszavai közt keresgélhetünk (a kulcsszavak azok a szavak, amelyek a help-szövegekben kiemelve jelennek meg), a **T***opic search* (*Ctrl-F1*) funkcióval a BORLAND C++ abc-sorrendbe szedett "tárgymutatójában" kereshetünk meg egyes fogalmakat (akár C vagy C++ nyelvi elemeket, akár a fejlesztő rendszer használatával kapcsolatos dolgokat), a **P***revious topic* (*Alt-F1*) menüponttal a "tárgymutató" eggyel korábban megtekintett elemére térhetünk vissza, míg a **H***elp on help* funkció pedig magáról a help-rendszerről nyújt információkat. Egy tipikus help-et láthatunk az 1.3. ábrán.

2 Fejezet

C programozás a BORLAND C++-ban

A BORLAND C++ rendszer az ANSI szabványajánlásnak megfelelő, korszerű, sok hasznos szolgáltatást nyújtó C nyelvi implementáció. Ismertetésünk során igyekeztünk kerülni a nyelv formális definícióját, inkább szavakba öntve, példákon keresztül próbáltuk azt bemutatni. A nyelv bemutatását az ANSI C, illetve annak a BORLAND C++-beli implementációjának ismertetésével kezdjük. A 3. fejezet foglalkozik a nyelv objektum-orientált vonásainak a tárgyalásával. A 4. fejezetben rövid leírást találhatunk a BORLAND C++ IBM PC specifikus tulajdonságairól (overlay technika, képernyőkezelés). A BORLAND C++ rendszer könyvtári függvényeinek rövid leírását az A függelék tartalmazza.

Könyvünk második kötete azzal foglalkozik, milyen módon használható a BORLAND C++ rendszer MS-Windows alkalmazói programok fejlesztésére.

Ha az Olvasó az egyes C nyelvi elemek precíz leírása iránt érdeklődik, akkor a legelső és legfontosabb C nyelvi referenciát, B. W. Kernighan és D. M. Ritchie könyvének [1]. A függelékét ajánljuk tanulmányozásra. A nyelv BORLAND C++ implementácóval kapcsolatos részleteit az eredeti programdokumentáció vonatkozó kötetei [10], [11] tartalmazzák. Külön erénye a programdokumentáció ezen köteteinek, hogy kitérnek a portabilitási kérdésekre is.

Megadunk itt egy egyszerű kis példaprogramot, amelyre később többször is hivatkozni fogunk, javasolva, hogy az Olvasó lépjen be a a BORLAND C++ integrált fejlesztői környezetébe és gépelje be a programot, majd fordítsa le és próbálja is ki. A program funkciója, hogy *ENTER*-rel

lezárt sorokat kér be a felhasználótól, és azokat úgy írja vissza a képernyőre, hogy a kisbetűket a nagybetűs párjukkal, a nagybetűket pedig a megfelelő kisbetűvel helyettesíti. A programból kilépni *CTRL-Z* billentyűleütést tartalmazó sor beírásával lehet.

Meg kell jegyeznünk, hogy nyomdatechnikai okokból a szövegben található programlisták magyar nyelvű megjegyézseiből hiányoznak az ékezetek. Ezért a kellemetlenségért az Olvasó elnézését kérjük.

```
/*************************************************************
* File:          pelda.c                                     *
* Tartalom:      Kisbetu-nagybetu felcserelo mintaprogram    *
*************************************************************/
#include <stdio.h>
#include <ctype.h>
                    /* A modulban definialt fuggvenyek:      */
void main(void);
/* ========================================================== */
void main()
{
    register c;
    while ((c = getchar()) != EOF)
    {                       /* c-be olvasunk, file vegeig     */
        if (isupper(c))     /* Ha nagybetu, akkor....         */
        {
            c = tolower(c); /* .... kisbeture csereljuk,      */
        } else              /* .... egyebkent pedig ....      */
        {
            c = toupper(c); /* .... nagybeture csereljuk.     */
        } /* ..............  Az 'if' utasitas vege ......     */
        putchar(c);         /* A megvaltoztatott c-t kiirjuk  */
    } /* ................... A 'while' ciklus vege .......   */
} /* ....................... A 'main' blokk vege .........   */
```

2.1 Alapismeretek

A C programutasításokat kötetlen formátumban írhatjuk. Ez azt jelenti, hogy – az előfeldolgozónak szóló utasításokat kivéve, lásd ott – a C nyelv sorfüggetlen, azaz egy sorba több utasítás is írható, illetve egy utasítás több sorba is törhető. Általános érvényű szabály, hogy ahol egy szóköz állhat, ott tetszőleges számú – legalább egy – tetszőleges szóközjellegű karakter (ún. *whitespace*) is állhat. Szóközjellegű karakter a szóköz, a tabulátor és az

újsor karakter, illetve ilyen funkciójú a megjegyzés (*comment*) is. A megjegyzések /* karakterkombinációval kezdődő és */ kombinációval lezárt, tetszés szerinti karaktersorozatok. Ennek alapján a `pelda.c` forrásprogram alábbi két sorát

```
void main()
{
```

a következő formában is írhattuk volna:

```
          void      /* komment */
main()      {
```

Modulnak nevezünk egy önállóan fordítható forrásnyelvi egységet. Az operációs rendszer szintjén a modul önálló file-ként jelentkezik. A C forrásmodulokat hagyományosan `.c` kiterjesztésű file-okban helyezzük el, kivéve az ún. include file-okat (*header files*, illetve *fejlécfile-ok*), amelyek `.h` kiterjesztést kapnak. A C programok egy, vagy több modulból épülnek fel, a `pelda.c` egymodulos program. Ha egy többmodulos programban csak az egyik modulhoz tartozó forrásfile-t módosítjuk, akkor elegendő csupán azt újrafordítani, majd a többi modullal – és a megfelelő modulkönyvtárakkal – újraszerkeszteni. A különálló modulok forrásnyelvi szinten deklarációkon keresztül tartják egymással a kapcsolatot, amelyeket célszerűen include file-okba foglalunk.

A modulok fordításvezérlő utasításokból, változó- és függvény-deklarációkból/definíciókból állnak. Az egyes tárolási egységek lehetnek kódgenerálók (ezekbe helyezzük a végrehajtható utasításokat), és lehetnek tárterület foglalók (adatok, változók). A kódgeneráló tárolási egységeket a C nyelvben általánosan függvényeknek nevezzük (szemben más programozási nyelvekkel, ahol különbséget tesznek a *function* és a *procedure*, illetve *subroutine* között). Egy tárolási egység deklarációja közli a fordítóprogrammal az adott egység jellemzőit, de nem jár együtt tárterület foglalással, az máshol (esetleg más modulban) történik meg. A deklarációt követően az adott tárolási egységet a fordító az adott tulajdonságokkal rendelkező, létező egységnek tételezi fel (ha egy deklarációhoz sehol sem tartozik megfelelő definíció, akkor a szerkesztés alkalmával hibaüzenetet kapunk). Ugyanazt a tárolási egységet többször is deklarálhatjuk egy forrásállományban mindaddig, amíg az egymást követő deklarációk összhangban vannak egymással. Egy tárolási egység definíciójakor történik meg az adott egység tényleges elhelyezése. Tárterület-foglaló tárolási egység esetén lefoglalásra kerül a szükséges adatterület, míg kódgeneráló egység esetén az adott utasításokat fordítja le és helyezi el a fordító program. (Itt adjuk meg a programunk végrehajtható utasításait). Egy definíció értelemszerűen deklaráció

értékű is, kivéve a kódgeneráló-egységek (függvények) definícióit, melyeket nem tekint teljes értékű deklarációnak a C fordító. Ezért célszerű minden forrásmodul elején elhelyezni az adott modulban definiálandó függvények teljes deklarációját (`void main(void);` sor a `pelda.c`-ben), ez egyben jól felhasználható dokumentálásra is. Ugyanannak a tárolási egységnek természetesen – a programhoz tartozó összes modult figyelembe véve – kizárólag egy helyen lehet definíciója (ezért "illetlenség" include file-ba definíciót helyezni).

A felhasználó azonosítás céljából a tárolási egységekhez azonosítókat (*identifier*) rendel. A C-ben az azonosító betűvel kezdődő, betűvel és/vagy számjeggyel folytatódó karaktersorozat. A kis- és nagybetűk különbözőnek számítanak (`alfa`, `AlfA`). Betűnek tekintjük az angol ABC betűin (`a`–`z`, `A`–`Z`) kívül az aláhúzás karaktert (`'_'`) is, számjegyek a szokásos `0`–`9` karakterek. Az azonosítók hosszára nincs megkötés, de a BORLAND C++ fordító csak az első 32 karaktert veszi figyelembe (más érték is beállítható az Options menüben). Javasoljuk, hogy a tárolási egységek elnevezésére használjunk csupa kisbetűt tartalmazó azonosítókat, és ne kezdjük saját azonosítóinkat aláhúzás karakterrel.

2.2 A változók alaptípusai

Ebben a részben a tárterület foglaló tárolási egységekkel foglalkozunk részletesebben. A továbbiakban az azonosítóval ellátott tárterület foglaló egységeket röviden változóknak nevezzük. A változók típusa meghatározza az elfoglalt tárterület nagyságát, az operátorokhoz tartozó gépi utasítás(sorozat)okat, stb. A BORLAND C++ a következő, tovább már nem bontható, elemi típusokkal rendelkezik:

- karakter,
- egészek (többféle méret és ábrázolásmód),
- lebegőpontos (többféle pontosság),
- mutató (pointer).

Ezeket az alaptípusokat azután – a későbbiekben leírt módokon – építőkövekként felhasználhatjuk ún. származtatott típusok képzéséhez, illetve bonyolultabb aggregátumok (tömbök, struktúrák, unionok, stb.) felépítéséhez. A különféle egészeket és a karaktertípust összefoglaló néven sorszámozott típusnak nevezzük. (A sorszámozott típus összefoglaló elnevezést a magyar nyelvű C szakirodalom (például [1]), integrális típusnak (*integral type*) nevezi. A fogalmi hasonlóság, és a magyarosabb hangzás miatt

C-beli jelölés	Karakter	Magyarázat
'\a'	(BEL)	csengő karakter
'\b'	(BS)	visszaléptetés
'\f'	(FF)	lapdobás
'\n'	(LF)	újsor (soremelés)
'\r'	(CR)	kocsi vissza
'\t'	(HT)	vízszintes tabulátor (vagy másképpen: TAB)
'\v'	(VT)	függőleges tabulátor
'\\'	(\)	maga a backslash karakter
'\''	(')	egyszeres idézőjel (apostrophe)
'\"'	(")	dupla idézőjel
'\?'	(?)	kérdőjel
'\ooo'		a ooo oktális kódú katkter
'\xhhh'		a hhh hexadecimális kódú karakter

2.1. táblázat: Backslash-szekvenciákkal megadható karakterek

megtartottuk a Pascal nyelvvel kapcsolatban általánosan használt terminológiát, a sorszámozott típus elnevezést)

2.2.1 A karakter típus és a sztringek

A karakter típusú változó egy karakter tárolására alkalmas. A BORLAND C++ fordító – az MS-DOS operációs rendszerrel összhangban – a karaktereket az ASCII tábla szerint kódolja. A karakterállandók megadása egyszeres idézőjelek között történik, például 'A', '(', stb. Lehetőség van bizonyos nem látható karakterek ábrázolására a *backslash* karakter segítségével. Ezeket a 2.1. táblázatban foglatuk össze.

A karakterláncok (sztringek) a C nyelvben nem alaptípusok, hanem karaktertömbökben ábrázoljuk őket. Mivel egy sztring hossza dinamikusan változhat, ezért a lefoglalt tárterületből adott pillanatban csak bizonyos karakterek tartoznak ténylegesen a karakterlánchoz. A problémát úgy oldották fel, hogy a karakterkészlet egyik elemét kijelölték arra a speciális funkcióra, hogy a sztringek végét jelezze, ily módon tehát ez a karakter magának a sztringnek soha sem lehet része. Ez a speciális karakter a '\0', azaz a 0 kódú karakter, amit a továbbikban az EOS szimbolikus néven (End Of String) fogunk hívni. A sztringállandókat a fordítónak dupla idézőjelek között lehet megadni, például "Hello". Ezt a fordító úgy értelmezi, hogy le

kell foglalni 5+1 karakternyi (6 byte) helyet a memóriában, és a következő karakterekkel, mint kezdőértékkel kell feltölteni: `'H'`, `'e'`, `'l'`, `'l'`, `'o'`, `EOS`. A sztringben lehetnek *backslash*-t alkalmazó szekvenciák is, például

```
"Ez egy idezojel: \"\n"
```

ami a memóriába a következőképpen kerül: `'E'`, `'z'`, `' '`, `'e'`, `'g'`, `'y'`, `' '`,`'i'`, `'d'`, `'e'`, `'z'`, `'o'`, `'j'`, `'e'`, `'l'`, `':'`, `' '`, `'"'`, LF, `EOS`. A sztringkonstansok nem nyúlhatnak át a következő sorba, de lehet őket egymás után való írással egyesíteni (konkatenálni), például:

```
"Ez egy 2 sorba irt "
"sztringkonstans"
```

2.2.2 Az egész típusok

Az egész típusú változók méret és értelmezés szerint többfélék lehetnek. Méret szerint a BORLAND C++ megkülönböztet rövid (`short int`) és hosszú (`long int`) egészeket, az előbbieket 16, az utóbbiakat 32 biten ábrázolja. Ha nem specifikáljuk az egész típusú változó méretét, hanem csak `int`-et adunk meg, akkor minden C fordító többé-kevésbé önkényesen választja azt meg, a BORLAND C++ például 16 bites hosszat vesz fel. Ha ki akarjuk használni a C nyelvben rejlő portabilitási képességet, akkor mindig használjunk explicit méretmegadást, az alábbi négy eset kivételével:

- `register` tárolási osztályú egészeknél,
- függvény által visszaadott egészeknél,
- egész típusú függvényparaméter definíciójánál,
- ha egy könyvtári függvény meghatározatlan méretű egészre mutató pointert vár. Ezeket a függvényhívásokat minden új implementációnál gondosan meg kell vizsgálni, összevetve a figyelembe veendő C rendszer deklarációival.

Az előbbi kivételek magyarázatát természetesen megadjuk a megfelelő helyeken. Ezekben az esetekben azonban a programozó felelőssége, hogy a méretmegadás nélkül felvett változót lgfeljebb 16 bit hosszúként kezelje (ennél kisebb hosszat `int`-re egyetlen komoly C fordító sem vesz fel).

Értelmezés szerint az egészek lehetnek előjelesek (`signed int`) és előjel nélküliek (`unsigned int`). Ha nem írjuk elő az értelmezést, akkor a fordító előjeles egészt vesz fel. Ily módon például a `short int` típusú változók -32768 és 32767 között, az `unsigned short int` típusúak pedig 0 és 65535

között vehetnek fel értéket. Itt jegyezzük meg, hogy a **char** típuson belül is létezik **unsigned char**, ez akkor lényeges, ha a szabványos ASCII készlettől eltérő értéket kívánunk benne tárolni. Ekkor a char úgy tekinthető, mint egy 8 bites egész típus, -128 és 127 illetve 0 és 255 közötti értékekkel. Fontos tudnivaló, hogy a BORLAND C++ a karakteres típussal való bármilyen műveletvégzés előtt (aritmetikai, logikai, összehasonlító, stb. művelet) azt 16 bitre terjeszti ki, figyelembe véve az értelmezését. Igy tehát bármilyen műveletvégzés szempontjából a **signed char**-ban tárolt hexadecimális 80 hexadecimális FF80-ként lesz figyelembe véve, míg az **unsigned char**-ban tárolt hexadecimális 80 ténylegesen hexadecimális 0080-nak számít. Tételezzük fel, hogy az **alfa** változó típusa **signed char**, a **beta** változóé pedig **unsigned char**, és mindkettőbe a 128 egész értéket írjuk. Ha **alfa**-t ezután összehasonlítjuk 128-cal, akkor hamis logikai értéket kapunk, míg **beta**-t összehasonlítva vele, igazat! Sőt, **alfa** sem lesz egyenlő **beta**-val.

Az egész típusú konstansokat decimálisan, oktálisan és hexadecimálisan is megadhatjuk. Az oktális konstansok kötelezően `0`-val kezdődnek, a hexadecimálisak pedig `0x`, vagy `0X` előtaggal (*prefix*-szel). Például `255` = `0377` = `0xFF`. A konstansokhoz is rendel méretet és értelmezést a fordítóprogram. A szabály az a BORLAND C++-ban, hogy a méret mindaddig 16 bit, amíg ennyi bittel a megadott szám ábrázolható, azon felül 32 bites az ábrázolás. Az értelmezés decimális konstansokra előjeles, a többire előjel nélküli. Például a `40000` előjeles egész konstans, mérete 32 bit (mert 16 biten a legnagyobb előjeles szám a 32767), de `0x89A0` előjel nélküli 16 bites konstans. Lehetőség van azonban az `l`, illetve az `L` utótaggal (*suffix*) előírni egy konstansra a 32 bites tárolást; az `u`, vagy `U` utótaggal pedig az előjel nélküli értelmezést. Például `0L` 32 bites előjeles, `40000U` pedig 16 bites előjel nélküli egész konstans.

2.2.3 A felsorolt típus

Szintén egész értékeket használ az ún. felsorolt (**enum**) típus is. Olyankor használjuk, mikor nem a konkrét számértékek a fontosak, csak az, hogy két érték egyenlő-e vagy sem, illetve melyik kisebb a másiknál. Ilyenkor elég felsorolni a használni kívánt egyedi azonosítókat, és a fordítóprogram automatikusan ellátja őket értékkel. Például az

```
enum szin { piros, kek, sarga, zold };
```

deklaráció létrehoz egy **szin** nevű felsorolt típust, amely a fenti egyedeket tartalmazza. Ha **alapszin** egy, a fenti típusba tartozó változó, akkor van értelme olyan értékadásnak, mint például

```
alapszin  = sarga,
```

és lehet olyan vizsgálatokat végezni, hogy az **alapszin** kék-e vagy sem, illetve nagyobb-e, mint kék (a nagyság szerinti sorrend megegyezik a felsorolási sorrenddel, a legkisebb van legelöl), azaz példánknál maradva

```
piros < kek < sarga < zold
```

A felsorolt típus a sorszámozott típusokkal kompatibilis típus.

2.2.4 Amit a logikai típusról tudni kell a C-ben

A C nyelvben nincs explicit logikai típus, erre a célra bármelyik sorszámozott alaptípus használható. A logikai igaz értéket az 1, a hamisat a 0 jelenti. Az operátorok közt vannak logikai jellegűek (például az összehasonlító operátorok), ezek mindig a fenti két érték valamelyikét szolgáltatják eredményül. Ugyanakkor azok az operátorok, illetve utasítások, amelyek logikai jellegű értéket várnak (például a feltételes elágaztatás), egy ennél tágabb értelmezést használnak: a 0 jelenti továbbra is a hamis logikai értéket, azonban minden más, nem 0 adatot igaznak fogadnak el.

2.2.5 A lebegőpontos valós számok típusai

A lebegőpontos valós számok a BORLAND C++-ban háromféle pontossággal állnak rendelkezésre. A `float` értékek 32 biten, a `double` számok 64 biten, míg a `long double` változók 80 biten tárolódnak. A pontosság mellett az ábrázolható abszolútértékek is változnak (lásd a 2.2. táblázatot). A lebegőpontos konstansok megadása a szokásos módon történik, a kitevő jelzésére `e` és `E` egyaránt használható.

2.2.6 A mutató típusok

A mutatók (pointerek) olyan változók, amelyek egyes tárolási egységek (változók vagy függvények) memória címeit tartalmazzák. Az ilyen változók tehát az adott tárolási egységre nem közvetlenül utalnak, hanem közvetetten (indirekció). A C nyelv ereje – többek között – éppen a mutatók kezelésében és a velük végzett műveletekben rejlik (lásd később a 2.9 részben). Ennek egyik forrása, hogy a pointer változókról nemcsak annyit tart nyilván a fordító, hogy ez egy mutató és hogy milyen méretű (16 vagy 32 bites), de azt is feljegyzi és követi, hogy a kérdéses változó vagy kifejezés milyen típusra mutat. Ezért valójában a C nyelvben nincs olyan alaptípus, hogy "mutató típus", csak olyan típusok vannak, mint például "mutató egy rövid egészre", "mutató egy előjel nélküli karakterre", vagy akár "mutató egy olyan mutatóra, amelyik egy `double` értékkel visszatérő, két előjel nélküli `long` paramétert váró függvényre mutat". Azt, hogy ez igazából nem is

bonyolult, azzal bizonyítjuk, hogy illusztrálásként megadunk egy definíciót az utolsó típusba tartozó **dfuncptrs** nevű változóra:

```
double (**dfuncptrs)(long, long);
```

A fenti példákból az is kitűnik, hogy a pointerek tárterület foglaló és kódgeneráló program egységekre egyaránt mutathatnak. Az előbbieket adatpointereknek, az utóbbiakat kódpointereknek nevezzük. A mutatók a BORLAND C++ rendszerben – illeszkedve a 8086-os mikroprocesszor felépítéséhez – 16, illetve 32 bit hosszúak lehetnek. A 16 bites mutatók neve **near** pointer, a 32 bitesek pedig további két csoportra oszlanak, a **far** pointerekre és a **huge** pointerekre. Egy **near** pointerrel legfeljebb 64 Kbyte tárterület címezhető meg, **far** és **huge** pointerrel a teljes 1 Mbyte-os adattár elérhető, de **far** pointerek esetében a megcímzett tárolási egység (például tömb) mérete nem lehet nagyobb, mint 64 Kbyte, míg a **huge** pointerekre semmilyen megkötés nincs. Ennek megfelelően az egyes mutatókkal való műveletvégzés bonyolultsága és végrehajtási ideje a fentiek szerint rohamosan nő. Azért, hogy a felhasználó az adott feladat igényei szerint optimális megoldást választhasson, a BORLAND C++ hat ún. memóriamodellt bocsát a felhasználók rendelkezésére. Ezek a következők:

- *tiny model*: mind az adat-, mind a kódpointerek **near** típusúak. Megkötés, hogy az összes adatnak és programkódnak el kell férni egy 64 Kbyte-os memóriaszegmensben. Az így fordított és szerkesztett programok **.com** típusúvá alakíthatók át.

- *small model*: mind az adat-, mind a kódpointerek **near** típusúak. Megkötés, hogy az összes adatnak el kell férni egy 64 Kbyte-os memóriaszegmensben, valamint az összes programkódnak el kell férni egy másik 64 Kbyte-os szegmensben.

- *medium model*: az adatpointerek **near**, a kódpointerek **far** típusúak. Megkötés, hogy az összes adatnak el kell férni egy 64 Kbyte-os memóriaszegmensben.

- *compact model*: az adatpointerek **far**, a kódpointerek **near** típusúak. Megkötés, hogy az összes programkódnak el kell férni egy 64 Kbyte-os memóriaszegmensben, valamint, hogy az összes statikus adatnak el kell férni egy másik 64 Kbyte-os szegmensben.

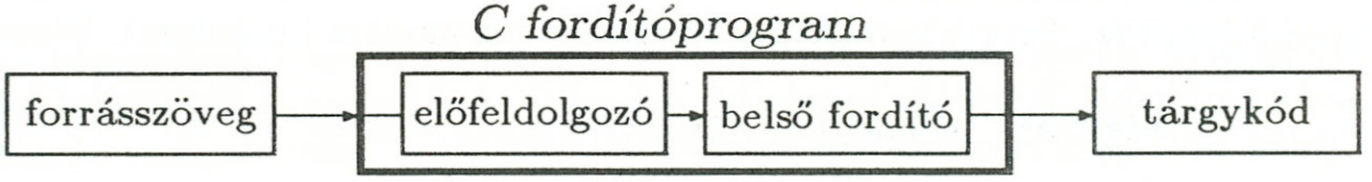
- *large model*: mind az adat-, mind a kódpointerek **far** típusúak. Megkötés, hogy az összes statikus adatnak el kell férni egy 64 Kbyte-os memóriaszegmensben.

- *huge model*: mind az adat-, mind a kódpointerek **far** típusúak.

Típus	Méret (bitben)	Értéktartomány	Pontosság (decimális jegyben)
`unsigned`	8	0 .. 255	-
`char`	8	−128 .. 127	-
`enum`	16	−32768 .. 32767	-
`unsigned int`	16	0 .. 65535	-
`short int`	16	−32768 .. 32767	-
`int`	16	−32768 .. 32767	-
`unsigned long`	32	0 .. 4294967295	-
`long`	32	−2147483648 .. 2147483647	-
`float`	32	$3.4 \cdot 10^{-38}$ .. $3.4 \cdot 10^{38}$	7
`double`	64	$1.7 \cdot 10^{-308}$ .. $1.7 \cdot 10^{308}$	15
`long double`	80	$3.4 \cdot 10^{-4932}$ .. $1.1 \cdot 10^{4932}$	19
`near` *pointer*	16	-	-
`far` *pointer*	32	-	-

2.2. táblázat: Adattípusok, méretek és értéktartományok a BORLAND C++-ban

Figyeljük meg, hogy a huge modell nem használ **huge** pointereket! A másik megjegyzésünk, hogy kódpointerben a **huge** típusra soha sincs szükség, mert egyéb megkötések miatt egyetlen függvény sem lehet 64 Kbyte-nál nagyobb. Ugyanez a megkötés vonatkozik a modulokra is, azaz sem az egy modulban definiált programkód, sem az ott definiált statikus adatterület nem haladhatja meg a 64 Kbyte-ot. A memóriamodell kiválasztását egy program írása során minden modul fordításakor célszerű ellenőrizni. Linkeléskor az integrált rendszer automatikusan gondoskodik róla, hogy a megfelelő modellű indító (ún. *startup*) kód és rendszerfüggvények legyenek beszerkesztve. Lehetőség van továbbá arra is, hogy tetszőleges memóriamodell használata esetén bizonyos mutatókat explicit módon **near**-nek, **far**-nak, vagy **huge**-nak deklaráljunk, sőt, ez az egyetlen lehetőség a **huge** pointerek használatára. A mutatókkal a 2.9-as részben foglakozunk részletesen.



2.1. ábra: Az előfeldolgozó kapcsolata a környezettel

2.3 Az előfeldolgozó

Minden C(++) fordítóprogram szerves részét képezi egy ún. *előfeldolgozó* (*preprocesszor*). 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. Ezt a 2.1. ábrán látható módon képzelhetjük el. Jellemző az ábrán látható együttműködésre, hogy a belső fordító nem ismeri a megjegyzések szintaxisát, ugyanis nincs rá szüksége: a neki átadott sorokból az előfeldolgozó azokat "kiírtja". A BORLAND C++ integrált fejesztői rendszerénél a két rész kapcsolata olyan szoros, hogy nincs is lehetőség az előfeldolgozó kimenetének a megtekintésére. Ehhez egy külön segédprogramot (`CPP` utility) találunk a programcsomagban. Ez szolgál arra, hogy a forrásszövegből előállítsa az előfeldolgozó kimenetét egy .i kiterjesztésű file-ba, így láthatóvá téve, hogy ténylegesen mi kerül át a belső fordítónak.

Az előfeldolgozó egy sororientált szövegfeldolgozó (más szóval makrónyelv), ami semmit sem "tud" a C nyelvről. Ez két fontos következménynyel 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 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.

2.3.1 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 program-

ban használt azonosítóktól, szokás szerint a szimbólumokat csupa nagybetűvel képezzük. Szimbólumokat a következő parancssal hozhatunk létre:

#define *szimbólum helyettesítendő szöveg*

A három fő részt tetszőleges számú, min. 1 db. szóköz és/vagy tabulátor választja el. Az előfeldolgozó minden beérkező sort átvizsgál, tartalmaz-e korábban definiált szimbólumot. Ha igen, akkor azt lecseréli a megfelelő helyettesítő karaktersorozatra, 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   TRUE      1
#define   FALSE     0
#define   YES       TRUE
#define   bool      int
#define   MENUCOL   20
#define   MENULINE  5
#define   BORDER    2
#define   MENUROW   (MENULINE+BORDER)
```

Az első három példában szereplő szimbólumokra szinte minden C programban szükség van. Figyeljük meg, hogy a **YES**-t a **TRUE** segítségével definiáltuk, ami kettős helyettesítéssel válik majd 1-gyé. A következőkben egy lehetőséget mutatunk arra, hogyan lehet a C nyelv alapszavait kibővíteni: bár explicit logikai típus a nyelvben nem létezik, mi létrehozhatunk egy új "alapszót", a **bool**-t, amit azután természetesen éppúgy használhatunk, mint az eredeti **int**-et (hiszen úgyis arra cserélődik le), de használatával az egyes változók szerepét jobban kidomboríthatjuk. Az alapszó jelleg hangsúlyozása érdekében használtunk ennél a szimbólumnál kisbetűket. A további definíciók a szimbólumok leggyakrabban használt területét mutatják be: legfőbb hasznuk az, hogy a különböző "bűvkonstansokat" 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   MENUSIZE  (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 makródefinícióra:

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

(Az itt szereplő, ún. feltételes kifejezéseket a megfelelő helyen tárgyalni fogjuk. Ha a ? előtt álló kifejezés logikai értéke igaz, akkor a ? és a : között álló, egyébként a : után álló kifejezés szolgáltatja az eredményt.) Figyeljük meg, hogy – szintén a szöveghelyettesítést szem előtt tartva – makrók esetében nemcsak a helyettesítő kifejezést, de a helyettesítendő 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. Itt hívjuk fel a figyelmet arra is, hogy **alfa** kiszámításához a fenti esetben az **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 **pelda.c**-ben használt **getchar()** is), ami a használatát nem befolyásolja – a fent említett ritka 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 telepítésénél 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 újradefiniálhatjuk ezt a szimbólumot is.

Megjegyzések

- A következő szimbólumok nem szerepelhetnek sem **#define**, sem **#undef** direktivákban:

  __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 esetleges szóközök). Az előfeldolgozó eltávolítja az esetleges szóközöket és a **##** jelet, és egyesíti a két különálló szintaktikai egységet. Ilyen módon lehet változókat konstruálni. Például a **#define VAR(I, J) (I ## J)** makródefinició 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 paraméterhivatkozások az **"arg"** alakkal helyettesítődnek. Például a

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

makródefinicó 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.

2.3.2 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 `#else` 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 értéke 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.

2.3.3 Előredefiniált szimbólumok

Szabványos szimbólumok

A BORLAND C++ mind az öt, az ANSI által megkívánt előredefiniált szimbólumot rendelkezésre bocsátja. Mindegyik szimbólum két-két aláhúzáskarakterrel kezdődik és végződik. Ezek a következők:

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

A BORLAND C++ saját előredefiniált szimbólumai

Az ANSI által megkívánt előredefiniált szimbólumokhoz hasonlóan ezek is két-két aláhúzáskarakterrel kezdődnek és végződnek.

- `__BCPLUSPLUS__` Csak C++ fordításkor van definiálva, értéke `0x0200`, a C++ fordító verziószámára utal. Ez a szám az újabb verziókban növekedhet.

- `__BORLANDC__` Hexadecimális állandó formájában megadja a BORLAND C/C++ fordító verziószámát (2.0-ás verzió esetén `0x0200`). Ez a szám az újabb verziókban növekedhet.

- `__TCPLUSPLUS__` Csak C++ fordításkor van definiálva, értéke `0x0200`, a C++ fordító verziószámára utal. Ez a szám az újabb verziókban növekedhet.

- `__TURBOC__` értéke a hexadecimális `0x0297` konstans. Ez a szám az újabb verziókban növekedhet.

- `__cplusplus` 1 értékkel van definiálva akkor, ha C++ üzemmódban használjuk a fordítóprogramot, egyébként definiálatlan. Ez lehetővé teszi, hogy egyes forrásmoduljainkat hol C, hol pedig C++ programként fordíthassuk.

- `__CDECL__` A BORLAND C/C++ fordító *Calling Convention ... C* opciójának bekapcsolt állapotában értéke 1, egyébként pedig definiálatlan. A következő szimbólumok közül minden esetben pontosan egy van definiálva (1 értékkel), a többi definiálatlan. A definiált szimbólum neve megegyezik a fordításkor alkalmazott memóriamodellel.

  ```
  __TINY__     __SMALL__   __MEDIUM__
  __COMPACT__  __LARGE__   __HUGE__
  ```

  Ha például az alkalmazott memóriamodell a *small*, akkor a `defined(__SMALL__)` értéke igaz (1), de a `defined(__LARGE__)` értéke hamis (0), s ugyanígy az összes többi szimbólum esetén.

- `__PASCAL__` Definiálva van 1 értékkel, ha az alapértelmezés szerint minden szimbólum a PASCAL konvenciót követi (a *Calling convention* fordításvezérlő kapcsoló *Pascal*-ra van állítva), egyébként definiálatlan.

- `__MSDOS__` A BORLAND C++-ban ez a szimbólum minden körülmények közt 1 értékkel definiálva van.

- `__OVERLAY__` definiálva van és értéke 1, ha egy modult ún. *overlay* modulként fordítunk.
- `_WINDOWS` definiálva van és értéke 1, ha egy programot MS-Windows alakalmazói programként fordítunk.
- `__DLL__` definiálva van és értéke 1, ha egy modult ún. *dynamic link library*-ként fordítunk le egy MS-Windows alkalmazáshoz.

2.3.4 File-beépítés

Igen gyakran használjuk a preprocesszor file-beépítési (*file include*) képességét. Segítségével teljes file-ok építhetők be a forrásunkba, pontosan úgy, mintha ott szerepelnének beírva. Az ilyen beépítendő include file-ok egy helyen összegyűjtve tartalmaznak szimbólum és makródefiníciókat, struktúrákat írnak le, más modulokban definiált tárolási egységeket deklarálnak stb. A BORLAND C++ rendszer nagyszámú include file-t bocsát a programozók rendelkezésére, amelyek közül az adott feladathoz szükségeseket építhetik be. A fejléc file-ok kiterjesztése hagyományosan `.h` (header file). A include file-okba beágyazva lehetnek újabb beépítési utasítások stb. A beépítési utasítás általános alakja:

`#include` <*file-név*>

illetve

`#include` "*file-név*"

ahol *file-név* a beépítendő file neve. Az első formát általában akkor alkalmazzuk, ha a rendszer által rendelkezésünkre bocsátott include file-t használunk (a `pelda.c`-ben is ily módon hivatkoztunk az `stdio.h`-ra), a másodikat pedig saját include file-jainkra.

2.3.5 Implementáció-függő vezérlősorok

Ezek 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++ három `pragma` utasítást

ismer. A `#pragma inline` vezérlősor csak a BORLAND C++ parancssor-változatában (`BCC`) használható , és arra figyelmezteti a fordítót, hogy beépített assembly utasítások találhatók a forrásszövegben. Ekkor a `BCC` nem tárgykódot generál, hanem assembly forrásszöveget, amelybe beiktatja a közvetlenül megadott assembly forrássorokat, és az így kapott szövegre elindítja a `TASM` assemblert. A második pragma-direktíva a `#pragma warn`. Ennek segítségével felül lehet bírálni a *Display warnings* fordításvezérlő kapcsolók beállított értékeit, és a forrásszöveg egyes részeire az alapértelmezéstől eltérő figyelmeztetési szintet választhatunk. A direktívának háromféle alakja van:

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

A fenti utasítások hatására az `xxx` figyelmeztető üzenet generálását bekapcsoljuk, az `yyy`-ét ki, míg a `zzz`-ét visszaállítja az eredeti értékére. A használható hárombetűs rövidítések és a hozzájuk tartozó figyelmeztetések jelentéseit a *BORLAND C++ User's Guide* 6. fejezetében találhatjuk meg.

A harmadik pragma-utasítás a `#pragma saveregs` direktíva, amely arra utasítja a fordítót, hogy egy `huge` függvényre olyan kódot generáljon, ami belépéskor elmenti az összes regiszter értékét, és kilépéskor visszaállítja azokat. A pragmát közvetlenül az adott függvény definíciója elé kell helyezni, és csak arra az egy függvényre vonatkozik.

2.4 Tárolási osztályok, hatáskörök

2.4.1 Modulok, blokkok.

Mint láttuk, a C programok 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, "elrejthetünk-e" változókat más modulok elől, stb. Ahhoz, hogy ezekre a kérdésekre válaszolni tudjunk, tovább kell boncolgatnunk a C programok szerkezetét. Említettük, hogy a modulokban vannak 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 alblokkja, és minden alblokk alá van rendelve az őt magában foglaló blokknak, míg a modulok azonos

hierarchiaszinten vannak, tehát egymás mellé vannak rendelve. Minden blokk két részre osztható: deklarációkra és adatdefiníciókra, valamint végrehajtható utasításokra (ez utóbbiak tartalmazhatják az alblokkokat). Egy blokk két része közül a deklarációs/definiciós rész el is maradhat. Mivel blokk belsejében nem lehet függvénydefiníció, ezért a C nyelvben minden kódgeneráló tárolási egység azonos hierarchia-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 meg a vezérlést. A C, illetve C++ nyelvben ennek a vezető függvénynek a neve kötelezően a `main` (lásd a 2.11-es részt).

2.4.2 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 tehát 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.

2.4.3 Változók élettartama és hatásköre

A tárterület 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 statikusaknak, az utóbbiba tartozókat dinamikusaknak, vagy automatikusoknak (`auto`) nevezzük. A statikus változók tehát már a program indulásának a pillanatában léteznek, ily módon – mint látni fogjuk – kezdőértékkel is rendelkezhetnek, míg a dinamikusaknak 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 me-

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`

2.3. táblázat: Változók élettartama, láthatósága és tárolási osztálya

móriával, hiszen az egyik blokkból kilépve, a felszabadított tárterület felhasználhatóvá válik a következő blokk számára, azaz a változók egy része a tárban – időeltolással – átfedheti egymást.

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 blokkra lokális változók hatásköre kiterjed az egész blokkra, beleértve az alblokkokat is, azonban ha egy ugyanolyan 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. Ezt a 2.3. 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. Ez nem kötelezi a fordítóprogramot semmire, és az, hogy mennyiben tudja teljesíteni igényünket, attól is függ, milyen típusú változóról van szó. Csak

sorszámozott, vagy pointer típusú változó lehet **register** tárolási osztályú. A regiszteres gépeknél az optimális tárolás általában valamelyik gépi regiszter használatát jelenti, ezért általában a gépi szó méretű változóknál képes a fordító teljesíteni a **register** ajánlást. 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 tehát az egyik olyan eset, amikor az **int** használata ajánlott. Az egész típusú változókon kívül **register** tárolási osztályú változónak ajánlhatók még a pointerek is.

2.4.4 Egyszerű adatdeklarátorok

Az 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 valamely blokk belsejében vagyunk, akkor a legbelső befoglaló blokkra nézve lokális, dinamikus változó jön létre (tehát **auto**, ezért ezt a kulcsszót nem is szokás sehol kiírni). Amennyiben a definícióból a típusmegadás hiányzik, akkor azt a fordító **int**-nek tételezi fel, ez akkor is igaz, ha típusmódosító kulcsszó (**short**, **long**, **unsigned**) mellett nem áll alaptípus. A **pelda.c**-ben a "**register c;**" definíció tehát egy előjeles értelmezésű, gépi szóhosszúságú egésznek foglal le helyet, amelyet dinamikus kezeléssel blokkra lokálisnak vettünk fel, és egyben kértük a fordítót, hogy igyekezzék optimális tárolást használni. (Ha valaki áttanulmányozza a lefordított kódot, láthatja, hogy valóban gépi regiszterbe is kerül.) Néhány további definíció:

```
long alfa;
static unsigned short beta, gamma;
double x;
signed char c;
unsigned char byte;
unsigned delta;
```

Ezek közül az első **alfa**-t hosszú előjeles egésznek definiálja (az **int** szokás szerint elmarad), és blokkon kívül globális hatáskörrel, blokkon belül **auto**-ként veszi fel. A **beta** és **gamma** egyaránt előjel nélküli rövid egész, statikus helyfoglalással; a definíció elhelyezése szerint modulra, vagy blokkra lokális láthatósággal. Az **x** duplapontosságú valós változó, **c** és **byte** karakteresek; előjeles, illetve előjel nélküli értelmezéssel, míg **delta** előjel nélküli

gépi szóhosszúságú egész. Ez utóbbi négy változó vagy globális, vagy **auto**, attól függően, hogy definícióik blokkon kívül, vagy belül vannak. Statikus helyfoglalású változó definíciójában előírhatunk kezdőértéket is úgy, hogy a definiált azonosító után egyenlőségjellel elválasztva kerül az inicializáló kifejezés, 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; ugyanakkor felhívja a figyelmet arra, hogy ki fogjuk ezt használni, és ha netán az **EOS** értékét megváltoztatnánk, akkor sem fogunk csalódni). 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. Emiatt az itt használt kifejezésre nincs semmilyen megkötés, még függvényhívásokat is tartalmazhat.

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 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-ben használni, amelyik az adott modulban – akár include file-okon keresztül – előzőleg deklarálva van. Egy modulra nézve külső globális változók deklarációja formailag megegyezik definíciójukkal, a tárolási osztályuk **extern**, és természetesen nem szerepelhet kezdőértékadás. 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];
```

sor egy 20 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, { }-ek közé zárt elemlistával, például:

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

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

```
static char hiba[ ] = "Hibas nev!";
```

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

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

```
unsigned short abc;
```

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

```
unsigned short *bcd;
```

definiálja a **bcd**-t, mint egy rövid, előjel nélküli egészre mutató pointert. Hasonlóan **register char *p;** definiálja **p**-t, mint egy (előjeles) karakterre mutató pointert. A **p** mutató dinamikus helyfoglalással kerül tárolásra, blokkra lokális és lehetőség szerint regiszterben lesz elhelyezve (ez utóbbi a BORLAND C++-ban függ a memóriamodelltől is). Hasonló elven tetszőleges alaptípusra mutató pointer definiálható. A tömbökre és a mutatókra később még részletesebben visszatérünk a 2.9-as alfejezetben.

2.4.5 Függvények definíciója és deklarációja

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

tárolási osztály típus azonosító(formális paraméterlista) blokk

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 függvény nem adhat vissza tömböt vagy függvényt, de visszaadhat ezeket megcímző mutatót). A függvényazonosítóra (*azonosító*) az azonosítókról általában elmondottak érvényesek. A ( )-ek között álló *formális paraméterlista* a paraméterek deklarációjára szolgál, ahány deklaráció szerepel benne, annyi paramétere van a függvénynek (akár egy sem, mint például a **main()** a **pelda.c**-ben). A blokk – { }-ek között – deklarációkat, blokkra lokális adatdefiníciókat és végrehajtható utasításokat tartalmazhat. Például a:

```
static double deg_to_rad(double x)
{
   ...
}
```

definiálja a **deg_to_rad** azonosítójú modulra lokális függvényt, amely egy dupla pontosságú lebegőpontos paraméterrel rendelkezik és ugyanilyen típusú értékkel tér vissza (a blokkban használható utasításokat a következő részekben fogjuk ismertetni). Vagy tekintsünk egy másik példát:

```
int read(int handle, char *buf, unsigned len)
{
   ...
}
```

Ez a definíció létrehozza a **read** azonosítójú globális függvényt, amely (előjeles) egész értéket ad vissza és három paramétert vár, ezek típusa sorrendben egész, karaktert megcímző pointer és előjel nélküli egész.

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** include file-ban deklarált

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

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

A fentebb ismertetett definíciós forma az ún. modern stílus szerinti, a BORLAND C++ által is elfogadott alak. A hagyományos forma (*classic style*) a következő:

tárolási osztály spec. típus azonosító(azonosítólista)
deklarációlista
blokk

Példaként álljon itt az előbb definiált **read** függvény hagyományos formátumú definíciója:

```
int read(handle, buf, len)
int      handle;
char     *buf;
unsigned len;
{
   ...
}
```

A hagyományos forma kicsit több gépelést kíván, talán kevésbé áttekinthető, de van egy nagy előnye: sokkal szélesebb körben használható, mint a modern változat, ugyanis a klasszikus stílusú függvénydefiníciókat minden (régi) C-fordító képes "megérteni", míg a modern stílust csak az újabb fordítók támogatják. Ugyanakkor, ha egyes, tisztán C nyelven írt forrásmoduljainkat esetleg C++ környezetben is szeretnénk használni, feltétlenül a modern deklarációs stílust kell alkalmaznunk. Ennek okára a 3. fejezetben az ún. *sokalakúság* kapcsán világítunk rá. Hogy melyik stílust használjuk a függvények definiálására, illetve deklarálására, a programozónak kell eldöntenie. Ha széles körben portábilis programot szeretnénk írni, célszerű a hagyományos stílus, míg ha a jövőbe tekintünk, és C++ modulokkal közös

project-ben szeretnénk hagyományos C modulokat is (átírás, modernizálás nélkül) felhasználni, a modern stílus alkalmazását javasoljuk. A kettőt nem mindig lehet keverni a BORLAND C++-ban. Ha C++ moduljaink is vannak a project-ben, akkor a BORLAND C++ fordító a hagyományos stílusú függvény definiciókat, illetve a formális paraméterlista nélküli deklarációkat nem fogadja el. A fenti dilemma feloldására célszerű a preprocesszor által nyújtotta feltételes fordítás lehetőségét kihasználni.

Mi e fejezet hátralévő részében a modern formát használjuk példáinkban. Természetesen, ha a portabilitás érdekében régi C fordítókra is fel kell készülnünk, akkor célszerű a hagyományos alakot használni.

A `pelda.c` mintaprogramot tanulmányozva feltűnhet, hogy a `main` függvény által visszaadott érték típusáról (`void`) eddig még nem tettünk említést. Ennek az az oka, hogy a `void` éppen azt jelenti, hogy nem létező, nem definiált típusú tárolási egység. Vannak függvények, amelyek szerepe az, hogy meghívásukkor bizonyos feladatot végrehajtsanak, de az általuk visszaadott érték közömbös, érdektelen (más programozási nyelvekben ezeket a programegységeket szubrutinnak vagy eljárásnak nevezik). A `void` "típusú" függvénydeklaráció azt jelzi a fordítóprogramnak, hogy az adott függvény nem ad vissza értéket, és így az figyelmeztethet, ha tévedésből mégis felhasználjuk a függvény értékét. (A `void` "típus" egy másik jellemző használata a `void*` típusú mutatók deklarálása. A `void*` típussal általános pointereket deklarálhatunk (lásd 2.9.7-et): azt jellezzük ezen a módon, hogy pontosan még nem tudjuk, milyen típusú tárolási egység címéről van szó, a pontos típus majd később derül ki, és akkor a pointer típusát a megfelelőre konvertáljuk – lásd később a típuskonverzió operátorát a 2.5.2 szakaszban.)

Ha egy más modulban definiált globális vagy könyvtári függvényt szeretnénk meghívni, függvénydeklarációt (*function prototype*) célszerű használni (sőt, a C++-ban ez egyenesen kötelező). A C nyelv megkötései ezen a téren nem olyan szigorúak, mint a változók esetén, ugyanis szabad meghívni nem deklarált függvényt, ha az `int` értéket ad vissza. Ezzel azonban kizárjuk azt a lehetőséget, hogy a fordítóprogram a paramétereket ellenőrizhesse. Javasolt ezért, hogy minden függvényt meghívása előtt deklaráljunk. A szabványos könyvtári függvények deklarációi mind megtalálhatók a megfelelő include file-okban, tehát ezek beépítésével a deklarációk automatikusan érvényre jutnak. A leggyakrabban használt `stdio.h` tartalmazza a főbb be- és kiviteli függvények deklarációit (ezenkívül fontos struktúra- és szimbólum-definíciókat); a `string.h` a karakterláncok kezelését végző függvényekét; az `stdlib.h` az általános jellegű, a `math.h` a matematikai függvények deklarációit tartalmazza, egyéb hasznos definíciók mellett; stb.

A függvénydeklaráció formája nagyon hasonlít a definícióéra:

tárolási osztály típus azonosító(formális paraméterlista);

Látható, hogy elmarad a függvénytörzset alkotó blokk, és megjelenik a lezáró pontosvessző (;). A tárolási osztály lehet **extern**, vagy **static**, ha elmarad, az **extern**-t jelent. A **read** deklarációja – a modern stílusú definiciójához hasonlóan – tehát a következő:

```
extern int read(int handle, char *buf, unsigned len);
```

A függvénydeklaráció másik – a fentivel egyenértékű, de szélesebb körben (azaz régebbi C fordítóknál is) alkalmazható – formája:

tárolási osztály típus azonosító(típuslista);

A **read** példájánál maradva:

```
extern int read(int, char *, unsigned);
```

A paraméterek helyén szereplő típusdeklarációkat úgy kapjuk, hogy a teljes paraméter deklarációkból elhagyjuk az azonosítókat.

Lehetőség van arra is, hogy a teljes paraméterdeklarációt elhagyjuk, ekkor csupán azt közöljük a fordítóval, hogy az adott azonosító függvénynév, és milyen típust ad vissza. (Ez a megoldás igazodik a klasszikus stílusú függvénydefiniciókhoz, ugyanis a régebbi C fordítók sokszor nem fogadják el a deklarációban a paraméterlista megadását.) Ez minimálisan szükséges, ha a függvény nem **int**-et ad vissza. Ha egy függvény nem vár paramétert, azt a deklarációban úgy jelezzük, hogy a paraméterlista helyére a **void** kulcsszót írjuk, mint ahogy ezt a **pelda.c**-ben is tettük.

2.4.6 Módosító jelzők

Kernighan és Ritchie [1]-ben 3 módosító jelzőt definiál , ezek a **short**, a **long** és az **unsigned**. A BORLAND C++ az ANSI szabványajánlásnak megfelelően további hármat valósít meg, ezek a **signed**, **const** és **volatile**.

A signed módosító jelző

A **signed** jelző azt állítja egy adott objektumról, hogy nem **unsigned**, tehát 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**-et 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 int a    = 100;
const int b[ ] = { 1, 2, 3, 4 };
```

Az első definició egy **a** nevű egész konstanst hoz létre, melynek értéke 100, a második definició pedig egy **b** azonosítójú, 4 elemű egész konstansokat tartalmazó tömböt eredményez. Mivel egy **const**-ként definiált változónak a programfutás során érték nem adható, mindig inicializáltan kell definiálni! 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 ointeren keresztül az adott sztringet soha se lehessen felülírni.

A volatile módosító jelző

Az ANSI C a **volatile** módosító jelzőt ugyan definiálja, de nem adja meg a komkrét, implementáció-független értelmezését. Egy **volatile** objektummal kapcsolatban csak azt a kívánalmat fogalmazza meg a C szabvány, hogy egy ilyen tárolási egységre vonatkozólag az adott C fordító semmiféle optimalizálást ne hajtson végre.

A BORLAND C++ esetében ez azt jelenti, hogy a a **volatile** jelző majdnem az ellentéte a **const**-nak. Azt jelzi, hogy a tárolási egység elvileg bármelyik pillanatban módosulhat egy, az adott programtól független

folyamat (például megszakítási rutin, DMA stb.) által. A fordítóprogram a **volatile** jelző hatására az adott változót nem fogja regiszterben tárolni, és az optimalizálás során sem fogja a többszörös, vagy látszólag felesleges értékadásokat kiiktatni. A **volatile** módosító jelző használatára tekintsük a következő példát:

```
volatile int ticks;

void interrupt timer(void)
{
   ticks++;
} /* end timer() */

void wait(int interval)
{
   ticks = 0;
   while (ticks < interval)
    ;
} /* end wait() */
```

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

2.4.7 Típusdefiniáló (typedef) azonosítók

A C nyelvnek van egy speciális "tárolási osztályt" jelző kulcsszava, ez a **typedef**. Ha egy azonosító definíciója mellé ezt a tárolási osztályt adjuk meg, akkor az azt jelenti, hogy valójában nem hozunk létre új dolgot, hanem az adott azonosítót – mint egy szimbólumot – a megadott típus megnevezésére kívánjuk használni. Ez tehát kizárólag a fordítónak szól, a generált kódban ennek semmi nyoma nem lesz. Például:

```
typedef int   bool;
typedef float float_array[20];
typedef short *short_ptr;
```

Az első példa ugyanazt a funkciót látja el, mint az előfeldolgozónál bemutatott szimbólum-definíció. A második létrehoz egy **float_array** nevű új típust, aminek segítségével lebegőpontos tömböket lehet definiálni. Tehát itt a megnevezett új típus a **float_array**, amelyet a **float** elemi típusból hozunk létre egy ún. típusmódosító operátor segítségével. Ez az operátor a [ ] tömbtípust képző operátor. Az utolsó példa egy mutatótípust – a rövid egészre mutató pointert – lát el azonosítóval. Ezt az új típust a * mutatótípust képző operátorral állítottuk elő a **short** elemi típusból. Ez utóbbi két példában használt szerkezetre a későbbiekben még visszatérünk. A fenti sorok leírása után szerpelhetnek a következő alakú definíciók:

```
bool        yesno;
float_array t11;
short_ptr   shp;
```

illetve alkalmazhatjuk a következő függvénydeklarációt:

```
extern bool ext_fv(short_ptr, bool, float_array);
```

A C-ben természetesen nem csak tárterületfoglaló tárolási egységekre vonatkozó típusokat hozhatunk létre, hanem kódgeneráló tárolási egységek típusait is definálhatjuk, azaz különféle függvénytípusokat adhatunk meg. A fentiekhez hasonlóan, ez a típusdeklaráció is olyan, mintha egy **typedef** "tárolási osztályú" függvényt deklarálnánk. Például a

```
typedef double dbl_fv(double, int);
```

deklaráció értelmében az új típus neve **dbl_fv**. Ezt a típust a **double** elemi típusból hoztuk létre a ( ) függvénytípust képző operátorral. A fenti példa szerinti **dbl_fv** olyan duplapontosságú valós értéket visszaadó függvény típusának az azonosítója, amely paraméterként egy **double** és egy **int** értéket vár. Az így létrehozott új típusazonosítót felhasználva írhatjuk, hogy

```
dbl_fv power;
```

ami teljesen ekvivalens egy modern stílusú függvénydeklarációval. Deklarálunk tehát egy **power** azonosítójú tárolási egységet, amelynek a típusa **dbl_fv**. Bár formailag úgy néz ki, hogy ez nem pusztán deklaráció, hanem egy komplett definició, ne feledjük, hogy a kódegeneráló tárolási egységek esetében a teljes definicióhoz szükség van a függvény törzsére is!

A függvénydeklarációk fenti módját nem javasoljuk, hiszen rejtve marad az, hogy **power** valójában nem egy változó, hanem egy adott fajta függvény, mindazonáltal a **dbl_fv** típus deklarálása nem haszontalan. Segítségével könnyen deklarálhatunk függvényre mutató pointereket:

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

2.4. táblázat: Típusmódosító operátorok. A precedencia felülről lefelé csökken.

```
dbl_fv *fv_mutato;
```

A függvénekre mutató pointerek használatával a 2.9.8-as részben foglalkozunk részletesebben.

Az egyszerű ökölszabályon – azaz a **typedef** "tárolási osztályú" deklarációk megadásán – túlmenően, az alábbiak szerint foglalhatjuk össze precízen egy új típus definiálásának szabályait.

Új típust mindig valamilyen már meglévő típusból (elemi típusból, struktúrákból, 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 (( ), [ ], *). Természetesen, ha egy bonyolultabb típust akarunk deklarálni (például egészekre mutató pointert visszaadó függvények tömbjét), figyelembe kell vennünk a típusmódosító operátorok precedenciáját, és ennek megfelelően kell a típusdefiníció során zárójeleznünk. A típusmódosító operátorok közül a * prefix operátor (a módosítandó típus előtt áll), míg a ( ) és a [ ] operátorok postfix operátorok (a módosítandó típus után állnak). Az típusmódosító operátorokra vonatkozó ismereteket a 2.4 táblázatban foglaltuk össze.

Típusdefiniálásra vonatkozó példákat találhatunk még a 2.9.1-es pontnál. A 2.12.2-es pontban a típusdefiniálással kapcsolatban további részletkérdésekre térünk ki.

Megjegyzendő, hogy a típusmódosító operátorok a szó szorosan vett értelmében nem operátorok, hiszen nem a C programunk futása során történő tényleges műveletvégzésre adnak utasítást, hanem csak a fordító program működését befolyásolják: arról adnak információt, hogy egy adott típust hogy, s mint kell kezelni a fordítás hátralévő részében.

2.5 Kifejezések

A kifejezés (*expression*) a C nyelv egyik kulcseleme. Az eddigiekben találkozhattunk vele (igaz, kicsit korlátozottabb formában) az előfeldolgozó `#if` típusú utasításai feltételrészében, tömbindexeknél és a definíciók inicializáló részében. Azonban látni fogjuk majd, hogy 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 egyoparandusú (*unary*), kétoperandusú (*binary*) és háromoperandusú (*ternary*) operátor. Példa lehet az egyes csoportokra a negálás operátora (`-x`), a szorzás operátora (`x*y`) és a makróknál bemutatott feltételes operátor (`x < y ? x : y`). A következőkben először áttekintjük az elsődleges kifejezéseket.

2.5.1 Elsődleges kifejezések

Elsődleges kifejezés az azonosító – feltéve, hogy már teljeskörűen 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 a 2.4.7 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 2.4.7 pontban elmondottakra utalunk.) Ez a gondolkodásmód összhangban van azzal, hogy egy azonosító mindig elsődleges kifejezés. A függvénynév is elsődleges kifejezés: önmagában leírva definició

szerint a függvény címét jelenti. Ezzel mást nem lehet csinálni, mint értékül adni egy, az adott típusú függvényre mutató pointernek (lásd 2.9.8), vagy a függvényaktivizáló operátort alkalmazni rá, miáltal az adott függvénybeli programkód az aktuális függvényparaméterekkel lefut.

A függvényhívás paramétereiben szereplő minden **float** érték automatikusan **double**, minden **signed char**, vagy **signed short int** érték **int**, minden **unsigned short int**, vagy **unsigned char** érték **unsigned int** típusúvá konvertálódik. (Ez a fajta konverzió, mint majd látni fogjuk, ennél sokkal általánosabb a kifejezések kezelésénél). Itt válik világossá, hogy függvény paramétereit miért értelmes gépi szó méretű **int**-re választani (ha garantáltan elférnek 16 bitben), és hasonlóan, miért célszerűtlen egy paramétert **float** típusúnak deklarálni.

Elsődleges kifejezések továbbá a struktúrák és unionok (lásd később) mezőire vonatkozó kiválasztó operátorok (a . és a ->) alkalmazásával nyert olyan kifejezések, amelyeknél az operátor baloldalán elsődleges kifejezés, jobboldalán pedig azonosító áll. Ezeket részletesebben a 2.8-es pontban fogjuk ismertetni.

Elsődleges kifejezésre példaként tekintsük először a **pelda.c**-ben előfordulókat. Ezek a következők:

```
c, getchar(), (c = getchar()), EOF, isupper(c),
tolower(c), toupper(c), putchar(c)
```

Tekintsünk néhány példát az előző részben található deklarációkat érvényben levőnek tételezve fel:

```
read(0, p, 0x100), p, hiba[8], vektor, szerda
```

Ezek típusai sorrendben **int**, **char ***, **char**, **float[ ]**, **int**, **int**. (Típust úgy kell megadni, mintha deklarálnánk egy ilyen típusú változót, majd elhagyjuk belőle az azonosítót. Az így kapott szerkezetet absztrakt deklarátornak nevezzük.)

2.5.2 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 operátornak. 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. *operator overloading* kapcsán tekintjük át részletesen a 3. fejezetben.

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 jelölt osztási művelet mást jelent egész típusú operandusokra alkalmazva (maradékos egész osztás), mint lebegőpontosakra, és kevert operandusok esetén nem dönthető el, milyen algoritmust kell használni. Hasonló – bár nem ennyire látványos – a probléma különböző méretű operandusok esetén is. Nem lenne praktikus elvárás, hogy a C fordító minden elképzelhető operanduspárosra adjon eljárást, de nem lenne célravezető a teljes uniformizálás sem (mint például egyes BASIC rendszereknél, ahol minden műveletet lebegőpontosan hajtanak végre). Ezért a C nyelv tervezői úgy döntöttek, hogy a gépi szónál rövidebb operandusokat gépi szó méretűre bővítik, valamint száműzik a rövid lebegőpontos műveleteket. 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. A pontos szabályokat a 2.6-os szakasz tartalmazza, és a továbbiakban ezekre mint "szokásos aritmetikai konverziók"-ra fogunk utalni.

Az ismertetést az egyoperandusú operátorokkal kezdjük, majd folytatjuk a kétoperandusúakkal, és végül kitérünk a C egyetlen, háromoperandusú operátorára is.

Az egyoperandusú opertárok közé tartozik a [ ] indexelő, a ( ) függvénynyaktivizáló operátor, és az elsődleges kifejezést képző operátor, az egyszerű zárójelpár. Ezeket az előző, 2.5.1. pontban már ismertettük, itt csak a teljesség kedvért 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*). Ha az 'address of' operátor operandusának típusa T, akkor az eredmény típusa "mutató T-re" lesz. Például a korábbi deklarációkat figyelembe véve a `p = &hiba[3];` értékadás után `*p` értéke `'a'` lesz. Az 'address of' operátor nem alkalmazható **register** tárolási osztályú vál-

zókra.

Az egyoperandusú - (minusz) 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 bitben. Például `-40000u` értéke `25536` lesz. A BORLAND C++ megengedi az egyoperandusú + (plusz) operátor használatát: a fordítóprogram egyszerűen figyelmen kívül hagyja azt.

A logikai tagadás operátora a `!`, 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 operátorának jele `~` (a tilde karakter), ami a – kötelezően sorszámozott típusú – operandusán elvégzett szokásos aritmetikai konverziók után kapott bitminta 1-es komplemensét adja eredményül, például `~0xFFFE` értéke 1 lesz (16 bites gépen).

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. Ha `x` értéke 3, akkor az `y = ++x * 2` eredményeként az `x` felveszi a 4 értéket, `y` pedig 8 lesz. Analóg módon létezik egyoperandusú prefix dekrementáló operátor is, jele `--`. Ez operandusa értékét csökkenti 1-gyel, és ezt adja eredményül.

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. Ha `x` értéke 3, akkor az `y = x++ * 2` eredményeként az `x` felveszi a 4 értéket, `y` pedig 6 lesz. (Most már érthetővé válik a *C++* elnevezés szimbólikus jelentése: olyan *C*, ami *egy fokkal* jobb.) Analóg módon létezik egyoperandusú postfix dekrementáló operátor is, jele `--`. Ez operandusa értékét csökkenti 1-gyel, és eredményül a csökkentés előtti értéket szolgáltatja.

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

típus. Például `(int)3.14` értéke `3`, `(long)0` megegyezik `0L`-val, `(unsigned short)-1` értéke `65535`, `(long *)p` pedig egy 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. Igy alkíthatjuk át az általános – ismeretlen típusú változóra mutató – `void*` "típusú" mutatókat adott típusú mutatókká. Erre a későbbiekben egy példát is be fogunk muatatni.

A `sizeof` egyoperandusú operátor az operandusaként szereplő változó (vagy zárójelben álló típus) byte-okban megadott méretét szolgáltatja. Ennek segítségével lehet gépfüggetlen módon tárterületigényeket meghatározni. Például `sizeof(int)` eredménye a BORLAND C++-ban 2, és minden implementációnál igaz, hogy `sizeof(char)` értéke 1.

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

$$1 \equiv \texttt{sizeof(char)} \leq \texttt{sizeof(short)} \leq \texttt{sizeof(int)} \leq \texttt{sizeof(long)},$$

illetve

$$\texttt{sizeof(float)} \leq \texttt{sizeof(double)}$$

A konkrét méretek a nyelv adott implementációjától függenek. (A BORLAND C++-ra vonatkozó méret információkat a 2.2. táblázatban foglatuk össze.) Általában nem érdemes a fentieknél többet feltételezni az elemi típusok méreteiről. Például nem mindig igaz, hogy az egész típus elég nagy ahhoz, hogy pointerek is elférjenek benne (lásd a BORLAND C++ esetében a `near` és a `far`, illetve `huge` pointereket). A korábbi deklarátorokat figyelembe véve `sizeof(p)` értéke memóriamodelltől függően 2 vagy 4, `sizeof(hiba)` értéke 11, `sizeof(vektor[2])`-é pedig 4. A fenti példákból is látszik, hogy az implementáció-független, portábilis C, illetve C++ programozási stílus kialakításában a `sizeof` operátornak kulcsszerepe van.

A kétoperandusú * operátor a szorzás művelete, a / operátor pedig az osztásé. A szokásos aritmetikai konverziók megtörténnek. 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.

A kétoperandusú + és - operátor a szokásos összeadást, illetve kivonást végzi el, a szokásos aritmetikai konverziók után. Speciális esetként lehet mutató operandusuk is, ezt a pointerekkel foglalkozó 2.9.4. fejezetben tárgyaljuk majd részletesen.

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 jobbraléptetésnél a belépő bitek az eredeti érték előjelbitjével egyeznek meg). Például `2 << 3` értéke `16`, `0xFFFDu >> 2` eredménye `0x3FFFu`, míg `-3 >> 1` értéke a BORLAND C++ esetén `-1`.

A relációs és egyenlőség-vizsgáló operátorok a következők: kisebb <, nagyobb >, kisebb vagy egyenlő <=, nagyobb vagy egyenlő >=, egyenlő == és nem egyenlő !=. Értelmezésük a szokásos, eredményül `int` 1-et vagy 0-át adnak, attól függően, hogy a vizsgált feltétel teljesül-e vagy sem.

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

A logikai ÉS operátor (`&&`) és a logikai VAGY operátor (| |) viszont logikai értékű operandusokat vár (0 - hamis, nem 0 - igaz). Eredményül `int` 0-át vagy 1-et szolgáltatnak a megfelelő logikai művelet elvégzése után. Vegyük példaként azt az esetet, hogy `a` értéke 2, `b` értéke 1. Ekkor `a & b`, `a | b`, `a && b` és `a | | b` értéke rendre 0, 3, 1 és 1 lesz, mivel logikailag mindkét operandus igaz értékű.

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 szokásos 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.

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: `y = x`. Kiértékelésre kerül a jobboldali kifejezés (szükség szerint konvertálva a baloldal típusának megfelelően), majd beíródik a baloldalon meghatározott változóba, felülírva annak korábbi tartalmát. Az egyszerű értékadó operátor baloldalán álló kifejezést az angol terminológia alapján *balértéknek* (*lvalue*), a jobboldalán álló kifejezést pedig *jobbértéknek* (*rvalue*) nevezzük. A kifejezés eredménye és típusa az értékadás során átadott értéknek megfelelő. Példaként tekintsük a `pelda.c`-ben szereplő alábbi feltételt:

```
(c = getchar()) != EOF
```

Mivel a zárójelek közt álló kifejezés elsődleges kifejezés, ezért ennek kiértékelése történik legelőször. A `getchar()` függvény hívása által szolgáltatott visszatérési érték beíródik a `c` nevű változóba, és ugyanez az érték lesz a zárójelezett kifejezés értéke is, ami végül összehasonlításra kerül az `EOF` szimbólummal, 0-át eredményezve, ha megegyeznek, és 1-et, ha nem. A `c` változó tehát a feltétel kiértékelése közben mellékhatásként kapott értéket. A további értékadó operátorok a következők:

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

Látható, hogy mindegyik egy kétoperandusú operátorból és az értékadás jeléből tevődik össze. Egy *kif1 op= kif2* formájú kifejezés kiértékelését úgy tekinthetjük, 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` értéke 1, akkor az `x += 2` kifejezés értéke 3, és mellékhatásként `x` is 3 lesz.

A vesszőoperátor (*comma operator*) nevét jeléről, a , karakterről kapta. A vesszőoperátorral elválasztott kifejezéspár sorban egymás után kiértékelődik, és az eredmény értéke és típusa megegyezik a második (jobb oldali) 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, amelyek közül a középső értéke 5.

Az előzőekben felsorolt operátorok előfordulási sorrendje megegyezik a prioritásuk sorrendjével, az egyértelműség kedvéért azonban álljanak itt az egyes prioritási szintek csökkenő sorrendben a hozzájuk tartozó asszociativitási iránnyal együtt:

1. Elsődleges kifejezések. Az elsődleges kifejezések balról jobbra csoportosítanak, tehát például `a.k[3]` értelmezése `(a.k)[3]`. Itt a `.` (a pont karakter) az ún. mezőkiválasztó operátor. Erről részletesen az struktúrákkal foglakozó 2.8. pontban szólunk.

2. Egyoperandusú operátorok: `*`, `&`, `-`, `+`, `!`, `~`, `++`, `--`, típusmódosítás, `sizeof`
   Az egyoperandusú operátorok csoportosítása jobbról balra történik, azaz például `*p++` értelmezése `*(p++)`.

3. Multiplikatív operátorok: `*`, `/`, `%`
   A multiplikatív operátorok balról jobbra csoportosítanak.

4. Additív operátorok: `+`, `-`
   Az additív operátorok balról jobbra csoportosítanak.

5. Biteltoló (shift) operátorok: `<<`, `>>`
   A Biteltoló operátorok balról jobbra csoportosítanak.

6. Relációs operátorok: `<`, `<=`, `>`, `>=`
   A relációs operátorok balról jobbra csoportosítanak.

7. Egyenlőség-vizsgáló operátorok: `==`, `!=`
   Az egyenlőség-vizsgáló operátorok balról jobbra csoportosítanak.

8. Bitenkénti ÉS operátor: `&`
   Lásd a felsorolás utáni 1. és 3. megjegyzést.

9. Bitenkénti kizáró VAGY operátor: `^`
   Lásd a felsorolás utáni 1. megjegyzést.

10. Bitenkénti VAGY operátor: `|`
    Lásd a felsorolás utáni 1. megjegyzést.

11. Logikai ÉS operátor: `&&`
    Lásd a felsorolás utáni 2. megjegyzést.

12. Logikai VAGY operátor: `||`
    Lásd a felsorolás utáni 2. megjegyzést.

13. Feltételes operátor: ? :
A feltételes kifejezések balról jobbra csoportosítanak.

14. Értékadó operátorok: =, +=, -=, *=, /=, %=, >>=, <<=, &=, ^=, |=
Az értékadó operátorok jobbról balra csoportosítanak, azaz például `x = y = 1` hatására mind `x`, mind `y` az 1 értéket veszi fel.

15. Vesszőoperátor: ,
A vesszőoperátor garantálja a balról jobbra való sorrendben történő kiértékelést.

1. Megjegyzés: 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 megválasztani. 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 a második tag kiértékelése történik előbb, azaz az indexeléshez az eredeti vagy a megnövelt `a`-t használjuk. Ugyanez a helyzet áll fenn a függvényhívások paramétereinél, ugyanis ezek kiértékelése sem kötött sorrendű. Például a `fugg(i++, a[i])` gépfüggően választja meg adott `i` érték mellett az átadandó tömbelemet.

2. Megjegyzés: 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 operandusuk alapján az eredmény nem egyértelmű. Például `a && b` esetében `b` kiértékelésére 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. Megjegyzés: 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. Megjegyzés: a feltételes operátor *kif1* feltételrészében, és minden egyéb helyen, ahol feltételt kell megadnunk, tetszőleges kifejezés szerepel-

het, 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 assignment* 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` alakba.

2.6 Konverziók

A BORLAND C++ a standard eljárásokat alkalmazza az egyes adattípusok közötti automatikus konverziókra. A következő pontok az implementációfüggő részeket ill. az alkalmazott bővítéseket ismertetik.

2.6.1 A 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 előjelkiterjesztés történik, kivéve, ha a karakterekre vonatkozó alapértelmezést előjel nélkülire állítottuk (*Default Char Type* – `unsigned`). A `signed char` típusú tárolási egységekre mindig előjelkiterjesztés történik, az `unsigned char` típusúak pedig mindig 0 értékű felső byte-tal lesznek 16 bitre bővítve.

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.

2.6.2 Konverzió mutatók között

A BORLAND C++-ban a különböző pointerek különböző méretűek lehetnek, a memória modelltől ill. a pointer explicit méretmegadásától függően (lásd `near`, `far`, `huge`, `_cs`, `_ds`, `_es`, `_ss`).

Minden mutató deklarálásakor meg kell adni a mutatott típust, ami lehet `void`, azaz bármi is. Deklarálásuk után a pointerekhez hozzárendelhetünk azonban tetszőleges más típusú mutatókat is. Az ilyen értékadásokra

a BORLAND C++ figyelmeztető üzenetet ad, kivéve, ha a deklarált típus a `void*`. Az egyetlen megkötés, hogy adatpointerek és kódpointerek között nem lehet konverziót végrehajtani.

2.6.3 Aritmetikai konverziók

A Kernighan és Ritchie szokásos aritmetikai konverziónak nevezi azt az eljárást, amely megadja, hogy a különböző típusú operandusok milyen konverzión mennek keresztül az aritmetikai operátorokkal alkotott kifejezésekben. A következőkben leírjuk a BORLAND C++ által alkalmazott algoritmus lépéseit:

1. A nem egész, az `int`-nél rövidebb illetve a szimpla pontosságú lebegőpontos értékek a 2.5. táblázatnak megfelelően egésszé illetve `double` típusúvá konvertálódnak. 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`-lá alakul.
3. Ha az egyik operandus `double`, a másik is `double`-lá alakul.
4. Egyébként, ha valamelyik operandus `unsigned long`, a másik operandus is `unsigned long` lesz.
5. Egyébként, ha valamelyik operandus `long`, a másik operandus is `long` lesz.
6. Egyébként, ha valamelyik operandus `unsigned`, a másik operandus is `unsigned` lesz.
7. Egyébként mindkét operandus típusa `int`.

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

2.7 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 vezérlőutasítások esetében, ahol ezt majd külön jelezzük. Minden utasítás viselhet cimkét, amely az utasítás elé írt azonosítóból és az elválasztó : karakterből áll.

A legegyszerűbb utasítás az üres utasítás: ; Leggyakoribb szerepe, hogy egyes ciklusok üres ciklustörzsét jelezze, valamint az, hogy cimkét viseljen a blokkzáró } karakter előtt.

TÍPUS	MIRE KONVERTÁLÓDIK	ELJÁRÁS
`char`	`int`	Általában előjelkiterjesztés
`unsigned char`	`int`	0 felső byte-tal bővítve
`signed char`	`int`	Mindig előjelkiterjesztés
`short`	`int`	Bitminta-másolás
`unsigned short`	`unsigned int`	Bitminta-másolás
`enum`	`int`	Bitminta-másolás
`float`	`double`	Mantisszát 0-ákkal bővítve

2.5. táblázat: A szokásos aritmetikai konverzióban alkalmazott eljárások

2.7.1 Kifejezés-utasítások

Bármely kifejezés pontosvesszővel lezárva szintén utasítás, és a kifejezés kiértékelését vonja maga után, majd a kapott érték eldobásra kerül. Természetesen ennek akkor van tényleges haszna, ha az adott kifejezésnek van valamilyen mellékhatása, mint például változó értékének módosítása vagy függvény meghívása. A `pelda.c`-ben az alábbi kifejezés-utasítások találhatók:

```
c = tolower(c);
c = toupper(c);
putchar(c);
```

Az első két kifejezésben két mellékhatást is kiváltottunk, egy függvényhívást és egy értékadást. (Valójában a fenti "függvények" mind makróként vannak megvalósítva, de ez mondanivalónk lényegét nem befolyásolja.) Szóba jöhetnek még a fenti típusúakon kívül olyan kifejezésutasítások is, amelyek inkrementáló, vagy dekrementáló operátort tartalmaznak, például `x++;` Ezekben az esetekben lényegtelen, hogy a prefix vagy a postfix változatot használjuk. Szeretnénk felhívni a figyelmet arra, hogy szintaktikailag a következő utasítások is helyesek:

```
hiba[3];
8;
a == b;
main;
```

de nem csinálnak semmit. (Ha engedélyezzük, a BORLAND C++ ezt is jelzi a *Code has no effect* figyelmeztetéssel.) Különösen a Pascal-os gyakorlattal rendelkezők figyelmét hívjuk fel nyomatékosan arra, hogy ha egy

függvény nem vár paramétert és nem is ad vissza (tehát teljesen *procedure* jellegű), meghívásakor akkor is ki kell mögé írni az üres ( ) zárójel párt, hiszen ez jelzi azt, hogy a függvényt aktivizálni szeretnénk (függvényaktivizáló operátor). A fenti utolsó példa is helyes szintaktikailag (az adott függvény kezdőcímét adja, lásd a 2.5.1 és 2.9.8 alatt leírtakat), és ha nem törődünk a figyelmeztetéssel (vagy nem engedélyezzük), nehezen kideríthető hibát szerzünk magunknak.

Bárhol, ahol egy utasítás állhat, szerepelhet összetett utasítás (blokk) is. A blokk formailag két részre osztható. Lehet deklarációs és blokkra lokális adatokat defininiáló része, valamint utasításokat tartalmazó része. Ez utóbbiba újabb alblokkok épülhetnek.

2.7.2 A feltételes utasítás

A feltételes utasításnak két formája létezik:

`if` (*kifejezés*) *utasítás vagy blokk*

illetve

`if` (*kifejezés*) *utasítás vagy blokk* `else` *utasítás vagy blokk*

Mindkét forma végrehajtása a *kifejezés* kiértékelésével kezdődik, és ha az igaz (nem 0), akkor az első utasítás vagy blokk kerül végrehajtásra. A második formánál megadott második utasításra vagy blokkra akkor kerül a vezérlés, ha a kifejezés hamis (0 értékű). A `pelda.c`-ben szereplő feltételes utasítás:

```
if (isupper(c))
{
   c = tolower(c);
}
else
{
   c = toupper(c);
}
```

esetén a feltétel – `isupper(c)` – azt adja meg, hogy a `c` változóban lévő karakter az angol ABC nagybetűje-e vagy sem. Ha a karakter nagybetű, akkor a `c = tolower(c);` utasítás segítségével a kisbetűs párjával helyettesítődik, ha nem, akkor a `c = toupper(c)` utasítás csinál a kisbetűkből megfelelő nagybetűket. Figyeljük meg, hogy bár mind az igaz, mind a hamis ág egyetlen utasításból áll, mégis blokkba foglaltuk őket. Ez nem kötelező,

de az a meggyőződésünk, hogy ezzel a stílussal olvashatóbb, hibavédettebb programokat írhatunk. Mindenki igazat fog nekünk adni, aki – hozzánk hasonlóan – csak egyszer is keresett ilyenfajta hibát:

```
if (isupper(c))
   c = tolower(c);
else
   c = toupper(); fugg(c);
```

ugyanis ez a programrészlet – az írásmóddal is jelölt – szándékunk ellenére a **fugg** függvényt minden esetben meghívja. A másik hasonló hiba:

```
if (a > b)
   if (a > amax)
      amax = a;
else
   if (b > bmax)
       bmax = b;
```

Az előbbi kódrészlet első ránézésre arra szolgál, hogy kiválasszuk **a** és **b** közül a nagyobbat, majd azzal – bizonyos feltételek tejesülése esetén – a megfelelő változót felülírjuk. A kódrészletet azonban a fordító így értelmezi:

```
if (a > b)
   if (a > amax)
      amax = a;
   else
      if (b > bmax)
         bmax = b;
```

ami teljesen mást jelent (például **bmax** csak úgy kaphat értéket, hogy **a>b** teljesül). Ennek oka az, hogy a fordító minden **else**-t a legközelebbi, őt megelőző, **else** nélküli **if**-hez kapcsolja. Ha azonban az általunk javasolt, blokkos formát alkalmazzuk

```
if (a > b)
{
   if (a > amax)
   {
      amax = a;
   } /* endif a>amax */
}
else
```

```
{
   if (b > bmax)
   {
      bmax = b;
   } /* endif b>bmax */
} /* endif a>b */
```

akkor a fenti hibákat nem követhetjük el. A blokkzáró } után álló megjegyzések összetett vezérlési szerkezetek esetén jelentenek hasznos fogódzkodót.

2.7.3 Ciklusszervező utasítások

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, stb. A ciklusból tehát mindaddig nem jutunk ki, amíg a ciklusfeltétel hamis nem lesz. A **pelda.c** program fő ciklusa is a **while** szerkezettel lett megoldva:

```
while ((c = getchar()) != EOF)
{
   ciklustörzs
} /* endw */
```

A ciklusban maradás feltétele itt az, hogy a terminálról beolvasott, és a **c** változóban tárolt érték ne a file-vége jel legyen. A fenti séma karakteres file-olvasásnál mindennapos a C-ben. Vegyük észre azt is, hogy ha a file nem tartalmaz (vagy a terminál nem küld) egyetlen karaktert sem, akkor a programunk illően nem csinál semmit. Gyakran előfordul, hogy a ciklus feltételrészében minden tennivalót elvégzünk, és a ciklus törzse üresen marad, például:

```
while ((c = str[i++]) == ' ')
  ;
```

Induláskor az **str** karakterlánc **i**-edik elemén állunk, és a ciklus végére **c** felveszi a sztring soron következő, nem szóköz karakterét, **i** pedig a karakterlánc rákövetkező elemének indexét tartalmazza.

Hátultesztelt, feltételvezérelt ciklusokat a következő formában írhatunk:

do *utasítás vagy blokk* **while**(*kifejezés*);

Az adott utasítás vagy blokk végrehajtása mindaddig ciklikusan ismétlődik, amíg a ciklusfeltételt jelentő kifejezés igaz (nem 0) marad. Akkor célszerű a használata, ha a ciklustörzset legalább egyszer mindenképpen végre kell hajtani.

Más nyelvekhez hasonlóan elsősorban a **for** utasítás szolgál arra, hogy számlálóvezérelt ciklusokat készítsünk; ugyanakkor a C nyelv **for** ciklusa jóval kötetlenebb. Általános alakja:

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

A ciklusba lépve kiértékelődik az *inicializáló_kifejezés*, ami valamilyen mellékhatáson keresztül inicializáló szerepet tölt be. Ezután a *feltétel_kifejezés* értékét határozza meg a gép, és ha értéke hamis (0), akkor elhagyjuk a ciklust. Egyébként végrehajtásra kerül a ciklus törzsét alkotó utasítás vagy blokk, majd a *léptető_kifejezés* értékének meghatározására kerül sor (ez egy mellékhatás révén a ciklusban történő léptetést vezérelheti), majd visszatérünk a ciklusfejben lévő *feltétel_kifejezés* kiértékelésére, és a ciklus mindaddig folytatódik, amíg *feltétel_fifejezés* hamis nem lesz. A fentiek alapján látható, hogy a **for** a **while** egy olyan kiterjesztéseként is felfogható, ahol a ciklus előtti inicializáló és a ciklustörzs legvégén szereplő léptető kifejezés-utasításokat egy helyre, a ciklusfejbe összevonjuk. Első példaként nézzünk meg egy, a hagyományos **for**-nak megfelelő számlálóvezérelt ciklust:

```
for (i = 0; i < dbszam; i++)
{
    ....

} /* endfor */
```

A fenti ciklus a törzsét **dbszam**-szor hajtja végre, illetve ha **dbszam** <= 0, akkor egyszer sem (azaz ez a változat is kicsit intelligensebb a hagyományos **for** utasításoknál).

A következő példa megkeresi a **float vektor[N]** tömbben az első negatív elemet:

```
for (i = 0; i < N && vektor[i] >= 0; i++)
```

Ha kilépéskor `i` értéke `N`, akkor minden elem nemnegatív, egyébként `i` a kérdéses elem indexe. A `for` ciklus fejében lévő kifejezések közül bármelyik el is maradhat. Ha a *feltétel_kifejezés*-t hagyjuk el, akkor az állandó igazat jelent. A

```
for (;;)
{
  ...
} /* endfor */
```

alakú ciklus a **végtelen ciklus**, kilépés belőle csak egyéb vezérlőutasításokkal lehetséges. (Egy másik lehetőség a végtelen ciklus létrehozására a `while(1) {...}` szerkezetű ciklus.)

2.7.4 Egyéb vezérlésátadó utasítások

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

`switch` (*kifejezés*) *blokk*

A kifejezés kiértékelésekor a szokásos aritmetikai konverziók megtörténnek, a kapott típusnak `int`-nek kell lenni. A blokkon belül bármely utasítás rendelkezhet a következő formátumú címkével (akár többel is):

`case` *állandó_kifejezés*:

Az *állandó_kifejezés* fordítási időben kiértékelhető kifejezést jelent, azaz nem függhet egyetlen változó értékétől sem. Nem használható két egyforma értékű `case` címke. A `switch` utasítás végrehajtása a kifejezés kiértékelésével kezdődik, majd az így nyert érték összehasonlításra kerül az összes `case` állandóval. Ha valamelyikkel megegyezik, akkor a vezérlés az adott `case` címkét követő utasításra adódik. Ha a kifejezés egyik `case` konstanssal sem egyezik meg, akkor a vezérlés a `default:` címkéjű utasításra adódik, ha ilyen létezik; ellenkező esetben a blokk egyetlen utasítása sem lesz végrehajtva. A blokk általában nem tartalmaz deklarációs illetve definíciós részt, ha mégis, akkor a definíciókban esetleg szereplő kezdőértékadások hatástalanok lesznek.

A `break;` utasítás befejezi az őt körülvevő legbelső `while`, `do`, `for`, vagy `switch` utasítás végrehajtását, és a vezérlés az így befejezett utasítás mögé kerül. A következő példa felhasználja a korábban definiált `enum nap` típusú `ma` változót:

```
switch (ma)
{
    case hetfo:

                        amit hétfő esetén csinálni kell

                        break;
    case kedd:
    case szerda:

                        keddi, vagy szerdai teendők

                        break;
    case szombat:

                        amit szombaton kell tenni

                        /* nincs break! */
    case vasarnap:

                        ami szombati vagy vasárnapi dolog

                        break;
    default:

                        amit egyébként csinálni kell

                        break;
} /* endswitch */
```

Figyeljük meg, hogy az egyes esetekhez tartozó utasításokat a következő **case** címke önmagában nem zárja le (egy címke a vezérlést nem változtathatja meg, csak célpont lehet egy vezérlésátadásnál), a tényleges lezárást a **break;** utasítással érhetjük el. Ki is használhatjuk ezt a tulajdonságot, mint a példánkban ezt **szombat** esetén tettük: a megfelelő esethez tartozó utasítások végrehajtása után "rácsorogtunk" a következő esetre. Természetesen az ilyen, nem nyilvánvaló megoldásokat feltűnően illik jelölni.

A **continue;** utasítás szolgál arra, hogy egy ciklus törzsének végrehajtását félbeszakítsa, és a vezérlést az adott ciklus fejére adja vissza (ez **while**-nál és **do**-nál a feltételvizsgálat, **for**-nál a léptetés). A **goto** utasítás a vezérlést a megnevezett címke utáni utasításra adja.

Egy függvény a hívójához a `return` utasítás segítségével térhet vissza. Az utasításnak kétféle alakja van:

```
return;
```

vagy

```
return kifejezés;
```

Az első formát a `void` típusú függvényeknél használjuk; a második forma esetén a függvény visszatérési értékét a kifejezés határozza meg. Szükség esetén az értékadó operátorhoz hasonló módon konvertálódik a kifejezés a függvény visszatérési típusára, de nem lesz `int`-nél rövidebb egész illetve `double`-nál rövidebb lebegőpontos típusú (ezért célszerű `int`-nek deklarálni a függvények visszatérési értékét `char` vagy `short` helyett). Megjegyezzük, hogy a függvény törzsét alkotó blokk végét jelző }-en való áthaladás egyenértékű a `return;` utasítás végrehajtásával.

2.8 Struktúrák és unionok

A korszerű programozási nyelvek a struktúrált programozást nemcsak vezérlési, hanem adatstruktúrák biztosításával is kötelesek támogatni, de amíg a vezérlési struktúráknak egy jól meghatározott köre van, addig az adatstruktúrák annyifélék lehetnek, ahány feladatra kell megoldást találni. Ezért új vezérlőutasítások létrehozását a programozási nyelvek ritkán támogatják, ellenben adatstruktúrák definiálását lehetővé kell tenniük. (Az objektumorientált programozás, így a C++ is még ezen a gondolatkörön is túllép: az összetett adatstruktúrákhoz a hozzájuk tartozó műveleteket is definiálva kapjuk az ún. *objektumokat*).

A C nyelvben az elemi típusokból felépített adatstruktúrákat aggregátumoknak nevezzük, és alapvetően háromfélék lehetnek: tömbök, struktúrák és unionok. Ez az elhatárolás azonban nem zárja ki ezek tetszőleges kombinációit, például szervezhetünk struktúra elemű tömbökből (és egyéb összetevőkből) új struktúrát vagy uniont.

Az eddig megismert egyetlen aggregátum, az egydimenziós tömb, a következő előnyöket biztosítja: tetszőleges eleméhez azonos idő alatt lehet hozzáférni (*random access*), mérete dinamikusan is beállítható (lásd később), 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 elemeket egységbe fogva kezelhessünk. Minden definiált adatstruktúra a fordító számára új típus-

ként 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 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 értékeként is.

2.8.1 Struktúrák megadása

A struktúramegadás formailag nagyon hasonlít az **enum** típusú deklarációkra, alakja:

tárolási osztály **struct** *típuscímke*
{
típus_1 azonosítólista_1;
típus_2 azonosítólista_2;
...
} *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. Példa struktúra megadására:

```
struct s1 {
            int       a, b;
            float     f;
            char      *s, nev[30];
            struct s1 *link;
          } sv, *s1h;
```

amely az **s1** struktúra alakjának rögzítése mellett definiál két – a definíció elhelyezésétől függően – globális vagy **auto** változót, az egyiket mint a fenti struktúrának megfelelő tárolási egységet, a másikat, mint egy ilyen típusú tárolási egységet megcímző mutatót. A struktúrát felépítő mezők típusa tetszőleges lehet, így bármely struktúra tartalmazhat újabb struktúrákat is (ezt beágyazásnak, angolul *nesting*-nek nevezzük); az egyetlen megkötés, hogy egy adott struktúrán belül elhelyezkedő elem típusa sem lehet a deklarálás alatt levő típus, azaz semelyik struktúra sem tartalmazhatja

önmagát saját elemeként. Lehetőség van viszont arra (amint az előbbi példában is látható), hogy a struktúra tartalmazzon mutatót olyan tárolási egységre, amelynek típusa az adott struktúra (lehetőség listák, fák és egyéb rekurzív adatstruktúrák kialakítására). A korszerű C implementációk – így a BORLAND C++ is – megengedik, hogy egy struktúraelem azonosítója megegyezzék valamely egyéb, más programegység azonosítójával, hiszen a mezőkre való hivatkozás – mint ahogy rögtön látni fogjuk – mindig egyértelműen megadja a megfelelő struktúrát is, így a két azonosító használata jól elkülöníthető. Ennek megfelelően lehet két különböző struktúrának is azonos nevű mezője. A struktúra típusú elemekből álló egydimenziós tömbök definiálása hasonlóan történik, mint az elemi típusokból való építkezés esetén:

```
stuct s1 s1_tomb[12];
```

A változódefiníciók struktúrák esetén szintén tartalmazhatnak kezdőértékadást is. A C nyelvben minden aggregátum inicializátorát – mint azt az egydimenziós tömböknél láttuk – a { } zárójelpár közé zárt listával kell megadni:

```
struct tanulo{
                  char nev[30];
                  short jegy;
                } osztaly[ ] =
                {
                  { "Nyilas Misi",   5 },
                  { "Pato Pali",     2 },
                      ...
                  { "Bolond Istok", 1 }
                };
```

A definiálandó struktúratömb méretét nem explicit módon írtuk elő, hanem az inicializáló elemek számával. A tömb kezdőértékét { } zárójelpár között levő listával adtuk meg – minden listaelem egy struktúra kezdőértékét határozza meg. Ezen aggregátumok inicializálása szintén { } közé zárt – kételemű – listákkal történik. Mivel karaktertömb alkotja a struktúra első mezőjét, ezért itt egyszerűsített megadást is használhattunk, sztringkonstans inicializátort alkalmazva. A második mező már nem aggregátum, ezért használhattunk az a kezdőértékadásban közönséges konstansokat.

2.8.2 Hivatkozás struktúra elemekre

Ha egy struktúra adott elemére kívánunk hivatkozni, akkor a . mezőkiválasztó operátort használhatjuk, például:

```
sv.link = svh; x = sv.f; osztaly[0].jegy = 5; (*svh).a = 0;
```

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

kifejezés->azonosító

forma teljesen megegyezik az alábbi alakkal:

*(*kifejezés) . azonosító*

Tehát a fenti utolsó példánkat a következőképpen is írhattuk volna:

```
svh->a = 0;
```

2.8.3 A bitmezők

A struktúrák egy másik felhasználási területe, hogy segítségükkel felbonthatunk sorszámozott típusú adatokat **char**-nál is rövidebb bithosszúságú részekre. Ily módon még arra is van lehetőségünk, hogy egy **int** értéknek minden egyes bitjét külön-külön kezelhessük. Az ilyen, bithosszban meghatározott elemek neve bitmező (*bitfield*), 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 egymástól 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:

tárolási osztály spec.`struct` *típuscímke*
{
típus_1 azonosító_1 : *hossz_1*;
típus_2 azonosító_2 : *hossz_2*;
...
} *azonosítólista*;

A bitmezők típusa `int`, vagy `unsigned int` lehet. 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 pakolja, egyébként újat kezd. Lehetnek meg nem nevezett bitmezők is, csak :-ból 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      mezo_1 : 1,
                             mezo_2 : 2;
                    unsigned mezo_3 : 8,
                             mezo_4 : 8;
                    int              : 0,
                             mezo_5 : 2;
                  } bm;
```

Egy szóba kerülnek a `mezo_1`, a `mezo_2` és a `mezo_3`. A `mezo_4` már nem fér itt el, ezért új szóban helyezkedik el. A `mezo_5` is elférne még ugyanitt, de a 0 szélességű bitmező használatával azt egy harmadik szóba kényszerítettük. Példák bitmezők használatára:

```
bm.mezo_3 = 'X';  bm.mezo_5 = bm.mezo_2;
bm.mezo_4 = bm.mezo_1 ? 0 : bm.mezo_3;
```

2.8.4 A union fogalma

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

```
struct fa {
            unsigned  jelzo_1 : 1,
```

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

Minden csomópontnak tartalmaznia kell a leszármazottaira mutató pointereket. A leveleket (vég-csomópontokat) arról lehet felismerni, hogy mindkét leszármazottuk hiányzik, tehát a jobb- és 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 a 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 a **union**. Példánkat segítségével ilyen formába írhatjuk:

```
struct fa {
        unsigned jelzo_1 : 1,
                 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 rész-típust külön-külön (de nem egyszerre) be tudja fogadni (A BORLANDC++ `near` mutatók esetén $2 \cdot 2$, `far` pointereknél $2 \cdot 4$ byte-ot fog lefoglalni). Ha `csp` típusa a fenti struktúra, akkor levél esetén írható például a következő értékadás:

```
csp.u.levelinfo = k;
```

míg ha a levélből csomópont lett

```
csp.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 struktúrákat a felsorolt elemek egyszerre, együttesen alkotják, míg a `union`-okban egyidőben a felsorolt elemek közül csak egy számára van hely. Fontos, hogy mindig a programozó felelőssége, hogy egy `union`-t mindig a benne aktuálisan tárolt adat típusának megfelelő módon használjon!

2.9 Mutatók és tömbök

A C nyelv egyik központi elemét képezik a mutatók. A pointer aritmetika következtében a C-ben 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 tipizáltsága 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 muatókkal kapcsolatos ismereteket.

2.9.1 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. Hasznos "ökölszabály" szintaktikailag a következő kijelentés: minden helyen, ahol adott tárolási egységet megnevező azonosító állhat, ott zárójelben állhat egy mutatóazonosító az indirekció operátorát (egyoperandusú *) követően. Például az `i` azonosító előfordulhat a következő környezetekben:

```
short i;
i = 8;
j = i + 2;
i++;
j = fugg(i);
```

tehát `i` helyére mondjuk `*ptr`-t írva a következő utasítások is helyesek szintaktikailag:

```
short (*ptr);
...
(*ptr) = 8;
j = (*ptr) + 2;
(*ptr)++;
j =  fugg((*ptr));
```

(természetesen a hivatkozott mutatót körülvevő ( ) zárójelpárok – az utolsó előtti utasítást kivéve – a precedenciaszabályok figyelembevételével elhagyhatók). Szemantikailag az első példa definiálja a `ptr` változót, mint egy rövid egészre mutató pointert. Ezt a deklarációt a zárójelek elhagyásával is írhatjuk: `short *ptr`.

Ezzel a deklarációval teljesen egyenértékű a 2.4.7-ból már ismert példa:

```
typedef short *short_ptr
...
short_ptr ptr;
```

Az előző példa második sorának értékadó utasítása a `ptr` által megcímzett, `short`-nak tekintett tárterületre beírja a 8 értéket. (Meg kell jegyeznünk, hogy ebben az esetben `ptr` valamilyen érvényes címet kell, hogy tartalmazzon, mert egyébként a `(*ptr) = 8` értékadás következménye beláthatatlan lesz. A példában szereplő ... helyére kell képzelnünk azt a müveletet, amelynek során a `ptr` pointerbe egy valamilyen módon lefoglalt, egyetlen `short` tárolására alkalmas memóriaterület címe kerül.)

A harmadik példában a mutatón keresztüli indirekcióval nyert értéket használjuk fel egy kifejezésben, a negyedikben megnövejük eggyel a rövid egésznek tekintett, a `ptr` által megcímzett adatot, míg az utolsó utasítás az indirekcióval nyert értéket adja át egy függvénynek. Kicsit bonyolultabb példák a bevezetőben említett "ökölszabály" alkalmazására:

a) `int func(double, int);` ⇒ `int (*pfunc)(double, int);`

Ez a példa azt illusztrálja, hogy deklarálhatunk egy függvényre mutató pointert. Esetünkben `pfunc` egy "`int` típusú értéket visszaadó,

egy **double** és egy **int** típusú paramétert váró függvényre mutató pointer" típusú változó. A **(*pfunc)** körüli zárójelek nem hagyhatók el, mert akkor **pfunc**-ot egy olyan függvénynek deklarálnánk (és valószínűleg sehol sem definiálnánk), amelyik egészre mutató pointert adna viszsza. Függvényre mutató pointerek esetén is célszerű a típust **typedef**-fel definiálni:

```
typedef int (*func_ptr)(double, int)
...
func_ptr pfunc;
```

Egy adott fajta függvényre mutató típust minden függvény-fajta esetében egyedileg kell "összeraknunk" . A típusmódosító operátorokban való gondolkodás itt is sokat segít: Az új típus a **func_ptr** lesz. Ezt az **int**-ből származtatjuk a következőképpen. A **(double, int)** postfix típusmódosító operátorral előállítunk egy – önmagában semmire sem használható – **int**-et visszadó függvény típust. Ebből a típusból a * operátor segítségével létrehozzuk a végső, típust: egy ilyen függvényt megcímző pointer típusát. Fontos a zárójelezés. Ha a fenti példában a ***func_ptr**-t nem védenénk zárójelekkel, akkor egy **int *** típusú visszatérési értékkel rendelkező függvénytípust definiálnánk. (A ( ) típusmódosító operátor precedenciája magasabb, mint a * típusmódosító operátoré, lásd a 2.4. táblázatot.) Természetesen érthetőbbé válik minden, ha a fenti típusdeklarációt részeire bontjuk:

```
typedef int    int_fv(double, int);
typedef int_fv *func_ptr;
```

b) **a = func(0.0, 81);** ⇒ **a = (*pfunc)(0.0, 81);**

Ezzel a példával azt igyekszünk megmutatni, hogy "ökölszabályunkat" alkalmazva hogy aktivizálhatunk indirekt módon függvényeket. A **pfunc** változó típusa az **a**) példa alapján (akár a **typedef**-es deklarációt, akár az anélküli deklarációt tekintve) "egész értéket visszaadó, egy dupla pontosságú valós, és egy egész típusú paramétert váró függvényre mutató" típus. Az indirekción alapuló függvényhívás szemantikája a következő: A függvényre mutató pointerre alkalmazzuk az indirekció operátorát, a *-ot. Ekkor – az általános szabályoknak megfelelően – a ***pfunc** kifejezés nem más, mint maga a tárolási

egység, mely esetünkben egy függvény. Egy "függvény típusú" kifejezéssel nem tudunk mit kezdeni, úgyhogy az indirekciót követően alkalmazzuk az függvényaktivizáló operátort, a `( )`-et (természetesen a szükséges függvényargumentumokkal együtt). Mivel ez utóbbi operátor precedenciája magasabb, mint az indirekció operátoré, `*pfunc` kifejezést zárójelekkel kell védenünk.

c) `float vektor[20]` ⇒ `float *(pvekt[20]);`

Itt `float` típusú tárolási egységekre mutató pointerek tömbjét deklaráltuk. A `short_ptr` deklarációjához hasonlóan egy kissé logikusabban átcsoportosítva az egyes típusmódosító operátorokat, `pvekt`-et így is deklarálhatjuk: `float* pvekt[20]` Ennek a példának a logikája nagyon hasonlít az a) pontban leírtakhoz. Jelen esetben a módosított `typedef`-es alak a következő lehet:

```
typedef float* float_vekt_ptr[20]
...
float_vekt_ptr pvekt;
```

vagyis a megnevezett új típus a `flaot_vekt_ptr`. Ezt a `float*` típusból származtatjuk úgy, hogy alkalmazzuk a postfix `[20]` tömbtípus képző operátort.

d) `x = vektor[2]` ⇒ `x = *(pvekt[2]);`

Most azt láthatjuk, hogy a c) pont szerint deklarált `pvekt` változó felhasználásával hogy érhetünk el egyes adatokat. Először a `[2]` indexelő operátorral előállítjuk `pvekt` megfelelő elemét (mely természetesen `float*` típusú), majd az indirekció operátorát alkalmazva megkapjuk a kívánt lebegőpontos értéket.

A `short *(*pptr);` példa "ökölszabályunk" rekurzivitását mutatja be, azaz egy mutató definíciójában lecserélve az azonosítót egy (itt feleslegesen) zárójelezett indirekt pointerre, sikerült egy mutatót megcímző pointert definiálnunk. Természetesen ezt tovább, tetszőleges szintig folytathatnánk, legalábbis elvileg.

Természetesen nagyon körültekintően kell eljárnunk az összetettebb típusú pointerek deklarálásakor. Ha a fenti c) példánál a zárójeleket máshova tesszük ki, egészen másfajta változó lesz a `pvekt`: a `float (*pvekt)[20]` deklaráció értelmében `pvekt` egy 20 elemű `float` típusú tömbre mutató pointer lesz.

2.9.2 Értékadás mutatóknak

Egy mutató értéket olyan értékadáson keresztül kaphat, ahol a jobb oldalon álló kifejezés típusa megegyezik a mutatóéval. Pointer típusú kifejezést legegyszerűbben az egyoperandusú **&** operátor (*address of* operátor) segítségével készíthetünk, vagyis konkrétan megadva azt, hogy a mutató milyen változóra mutasson, például (a 2.9.1 deklarációit feltételezve):

```
ptr = &i;
pptr = &ptr;
```

Speciális eset, ha függvény címét akarjuk beírni egy pointerbe. A fordító ilyen esetben nem igényli az **&** operátor alkalmazását (sőt, figyelmeztető üzenetet is ad érte), ugyanis definíció szerint egy függvényazonosító az őt követő paraméterzárójelek nélkül az adott függvény címét jelenti. Helyes tehát a következő értékadás:

```
pfunc = func;
```

Mutató típusú kifejezés előállításának másik módja az, hogy korábban értékkel feltöltött pointer(ek)ből megfelelő műveletekkel állítjuk elő. A pointereken végezhető aritmetikát rövidesen ismertetni fogjuk, itt csak legegyszerűbb esetként tekintsük a szimpla értékadást:

```
*pptr = ptr;
```

A leggyakrabban alkalmazott módszer mutatók feltöltésére a dinamikus területfoglalás során nyert címek felhasználása. A BORLAND C++ több standard könyvtári függvényt is rendelkezésünkre bocsát, amelyekkel futási időben lehet tárterület foglalást végezni. Ezek közül a legegyszerűbb a **malloc**, amely paraméterként a lefoglalni kívánt terület nagyságát várja (**sizeof** egységben). Az általa visszaadott érték (megfelelő típuskonverzió után) bármilyen pointernek átadható. Például a

```
ptr = (short *)malloc(sizeof(short));
```

hívás hatására futási időben a tárkezelő eljárás lefoglal egy akkora területet a memóriából, ahol egy rövid egész típusú adat elfér, majd az így kapott terület címe beíródik **ptr**-be. A lefoglalt területet azután a pointeren keresztül tetszés szerint manipulálhatjuk, éppúgy, mintha az eddig megszokott módon hoztunk volna létre egy **short** változót. A futási időben történő területfoglalás előnye, hogy pontosan az igényeknek megfelelő méretű területet vehetjük birtokba. Ennek elsősorban a dinamikus tömböknél (például karakterláncoknál) van nagy jelentősége.

Mutatóknak történő értékadásnál kiemelt jelentősége van a **NULL** szimbólumnak. (Ez a szimbólum többek között az **stdio.h** standard include

file-ban van definiálva). Az a mutató, amely **NULL** értéket kapott, úgy tekintendő, mint egy sehova sem mutató pointer, és a **NULL** garantáltan megkülönböztethető minden legális címtől. A `malloc` is **NULL** értékkel tér vissza, ha a területfoglalási kérelmet nem tudja teljesíteni. A **NULL** minden létező megvalósításnál a `0` érték (vagy inkább a `(void*)0` érték), de portabilitási és stiláris okokból előnyben kell részesíteni a **NULL** szimbólum használatát. A fenti területfoglaló példát is a következőképpen illik "tisztességesen" megvalósítani:

```
if ((ptr = (short *)malloc(sizeof(short))) == NULL)
{
  a memóriafoglalási hiba lekezelése
} /* endif */
```

2.9.3 Mutatók függvényparaméterként

Mutatókat igen gyakran használunk függvényhívásoknál. Ennek oka az, hogy a hagyományos C nyelvben minden, függvénynek átadott paraméter érték szerint kerül át. A függvénynek jogában áll paramétereinek értékét módosítani, de a módosítás nem hat vissza a paraméterként álló változóra. Ha az `f1` függvény definíciója a következő:

```
void f1(long a)
{
   a += 2L;
} /* end f1() */
```

akkor a függvény semmi hasznosat sem fog csinálni (a BORLAND C++ figyelmeztet is rá), azaz a következő kódrészlet után

```
alfa = 0L; f1(alfa);
```

`alfa` értéke nulla lesz. Hogyan lehet akkor olyan függvényt írni, amelyiknek kimenő vagy átmenő paraméterei is vannak? Természetesen a pointerek felhasználásával. Az adott mutató ugyan érték szerint kerül át, de az általa mutatott tárterület tartalmát a hívott függvény indirekció útján megváltoztathatja. Ilyen függvényt a pointerekre vonatkozó "ökölszabályunk" segítségével készíthetünk, azaz a függvényben a módosítandó paraméter minden előfordulását a zárójelezett indirekciós formával helyettesítjük:

```
void f1(long (*a))
{
   (*a) += 2L;
} /* end f1() */
```

és a hívásban is ennek megfelelően a változó helyébe annak címe kerül:

```
alfa = 0L; f1(&alfa);
```

A fenti kódrészlet után `alfa` 2-t fog tartalmazni.

A C++ az ún. *reference type* segítségével sokkal áttekinthetőbb megoldást nyújt a cím szerinti paraméterátadásra. Ezzel majd a 3.1.2-es pontban foglakozunk bővebben.

2.9.4 Pointer aritmetika

Pointerekkel különböző aritmetikai műveleteket végezhetünk, ezek a következő formájúak lehetnek:

- *pointer* + `int`,
- *pointer* - `int`,
- *pointer* - *pointer*,

illetve az első két művelet speciális eseteként

- ++*pointer*,
- --*pointer*,
- *pointer*++,
- *pointer*--

Minden mutatóval végzett művelet esetén képzeljük el azt, hogy a memória csak olyan típusú adatokból áll, mint amilyet az adott mutató megcímez, és ezek egyesével vannak megszámozva. Tehát például a `ptr` azonosítójú pointerrel való műveletvégzés esetén azt tételezzük fel, hogy a memória legkisebb megcímezhető egysége a rövid egészt tartalmazó szó. Ha most a mutatóhoz hozzáadunk, vagy belőle levonunk egy egész számot, akkor az a megadott számú adattal való előre-, vagy hátralépést fogja jelenteni, azaz például a `ptr+5` értéke az a cím lesz, ahol az aktuálisan mutatott `short` utáni 5-dik rövid egész szó elhelyezkedik; a `--ptr` pedig a megelőző `short`-ra állítja `ptr`-t. Más szóval, ha egy mutatott adat képzeletbeli sorszáma `i`, akkor a mutatóhoz való `n` egész érték hozzáadása után az eredményül kapott mutató az `i+n` sorszámú adatot fogja megcímezni. A fentieket legkönnyebb egydimenziós tömbök segítségével szemléltetni. Tekintsük az alábbi definíciókat:

```
float vekt[20], *pv = &vekt[4];
```

A **pv** azonosítójú mutatót egy adott adat címével inicializáltuk. Ez alkalmazható statikus helyfoglalású változókra is, mert csak azt kötöttük ki, hogy ezek inicializátorai nem függhetnek más változók értékétől, de itt nem az érték, hanem az elfoglalt cím lett felhasználva. A fenti definíciók után a **pv**, **pv + 3**, **pv - 2**, **pv + 20** kifejezések sorban a **vekt** tömb 5-ödik, 8-adik, 3-adik és – nem létező – 25-ödik elemének címét adják meg (az utolsó tehát szintaktikailag helyes ugyan, de szemantikailag hibás).

A *pointer* - *pointer* alakú kifejezésnek csak akkor van értelme, ha a két mutató által megcímzett típus azonos; ekkor az eredmény a két adat távolsága adat-méretnyi egységben, vagyis itt is alkalmazzuk a fenti feltételezést a memóriáról, és az eredményt a két mutató által kijelölt két adat sorszámainak különbségeként kapjuk. Példaként tekintsük az alábbi függvényt, amely egy karakterlánc hosszát adja vissza:

```
int strlen(s)
char *s;
{
   register char *p = s;
   while (*p != EOS)
   {
      p++;
   } /* endw */
   return (p - s);
} /* end strlen() */
```

A **p** mutatót ráállítjuk a sztring elejére, és mindaddig léptetjük előre egyesével (egy mutatott adattal, jelen esetben egy karakterrel), amíg el nem érjük a karakterlánc végét jelző **EOS** értéket. Eredményül éppen azt adjuk visszsza, hogy mennyit léptünk előre, amíg elértük a sztring végét, vagyis hány "igazi" karakter van a sztringben. A fenti szabványos könyvtári függvény egy felhasználási lehetősége:

```
char hiba[ ] = "Nincs eleg memoria!";
int  l1, l2;
...
l1 = strlen(hiba[0]);
l2 = strlen(&hiba[6]);
```

Ekkor **l1** értéke 19, **l2**-é 13 lesz.

A BORLAND C++ rendelkezik **huge** mutatókkal, amelyeknél lehetőség van arra, hogy ne csak **int** mennyiségeket adhassunk hozzájuk, illetve vonhassunk le belőlük, hanem használhatjuk a *pointer* + **long**, illetve *pointer* - **long** alakú műveleteket is. Ez nem portábilis lehetőség, ezért használatánál

erre legyünk figyelemmel. Hasonlóan, bár két mutató különbsége C nyelvi definíció szerint `int`, `huge` pointerek esetén ez az érték meghaladhatja a 16 bites határt. A fordító ezért a következő formát felismeri és 32 bites különbséget számol (`p`, `q` azonos típusú `huge` pointerek, `l` pedig `long`):

```
l = (long)(p - q);
```

A mutatók közötti bármiféle összehasonlítás (<, >, ==, stb.) csak akkor ad garantáltan helyes és portábilis eredményt, ha a mutatók azonos típusúak, és mindkettő egy, a mutatott típusból felépített tömbre mutat (lásd `strlen`-nél `p` és `q`, mindketten a `hiba[ ]` tömbre mutatnak). Hasonlóan garantált, hogy minden pointer biztonságosan összevethető `NULL`-lal, annak eldöntésére, hogy a mutató érvényes címet tartalmaz-e, vagy sem.

2.9.5 Tömbök használata. Többdimenziós tömbök

A C nyelv megengedi, hogy bármilyen adattípusból tömböt hozhassunk létre. Létezhetnek tehát struktúrákból álló tömbök, pointerek tömbjei, sőt, tömbökből alkotott tömbök is. Ezek deklarátorai illetve kifejezései a mutatóknál megismert "ökölszabály" analógiájára hozhatók létre: minden szinkaktiakailag helyes kifejezésben bármely azonosító lecserélhető egy

azonosító`[`*kifejezés*`]`

alakra – más szóval alkalmazzuk a `[ ]` tömbtípust képző típusmódosító operátort a deklarációnál. Például:

```
int i;                      int ti[4];
struct tanulo tan;          struct tanulo osztaly[40];
char *p;                    char *tp[10];
int (*pf)(int);             int (*tpf[5])(int);
float vekt[20];             float mat[5][20];
```

Az első két példa már ismerős, a következő kettő mutatókból álló tömbök deklarációját illusztrálja. A `tp` karaktermutatók 10 elemű tömbje, `tpf` pedig olyan pointerekből felépített 5 elemű tömb, amelyek egészeket visszaadó és egy egész paramétert váró függvényekre mutatnak. A fenti deklarációk szerint érvényes kifejezések a következők:

```
p = tp[2]
tp[0]
*(tp[8] + 1)
--tp[8]
pf = tpf[4]
(*tpf[2])(0)
```

Az utolsó példa többdimenziós tömbök definiálását mutatja be. A C fordító nem ismeri ugyan a többdimenziós tömböket, de a tömbök tömbjeit igen. Az alábbi **mat** változó egy 5 sorból és 20 oszlopból álló mátrixnak tekinthető:

```
float mat[5][20];
```

Ez például a következő kifejezésekben szerepelhet:

```
vekt[3] = mat[1][2]
mat[i][j] = 0.0
```

Többdimenziós mátrixoknak kezdőértéket adni a szokásos aggregátum inicializáló szintaxissal lehet:

```
short tomb[ ][3] = {
                     { 1, 2, 3 },
                     { 4, 5, 6 },
                     { 7, 8, 9 },
                     { -1, -2, -3 }
                   };
```

A **tomb** változó e definíció hatására egy 4 sorból és 3 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. Pontosan ez a többdimenziós tömbök legnagyobb hátránya, nem lehet őket dinamikusan kezelni. Nem írhatunk segítségükkel például olyan függvényt, amely egy $n \cdot r$-es mátrixot összeszoroz egy $r \cdot m$-essel, ahol a mátrixokon kívül n, r és m is paraméterek.

2.9.6 Kapcsolat tömbök és mutatók között

A következőkben a pointerek és tömbök közötti szoros kapcsolatot mutatjuk be. Vegyük elő az egyik fenti példát újra, kicsit más formában:

```
float 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 az 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 kifejezésekben 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 fenti **strlen()**-t alkalmazó példákat is írhattuk volna – és a későbbiek folyamán hasonló esetekben írjuk is – a következőképpen:

```
l1 = strlen(hiba);
l2 = strlen(hiba + 6);
```

Ebből viszont jól látszik az, hogy egydimenziós tömböt függvénynek paraméterként tulajdonképpen nem is lehet átadni, csak egy olyan mutatót adhatunk át, amely az adott tömb első (vagy bármelyik) elemére mutat. Hasonlóan, ha van a memóriában egy elegendően nagy terület, és arra mutat egy pointer, akkor az adott pointert úgy tekinthetjük, mintha tömb volna. Ez a mutatók és tömbök közti szoros kapcsolat teszi lehetővé, hogy

egydimenziós tömböket dinamikus indexhatárral kezelhessünk, azaz a méretüket elegendő futási időben rögzíteni. Például a fenti **vekt** használható a következő formában is:

```
float *vekt, *seged;
...
vekt = (float *)malloc(n * sizeof(float));
```

ahol **n** értékét elég futási időben meghatározni. Az így definiált **vekt** és az eredeti változat használata között semmi különbség nincs, csak az indexhatár más. A következő ciklus például kinullázza a tömböt:

```
int i, n = MERET;
...
for (i = 0; i < n; i++)
{
   vekt[i] = 0.0;
} /* endfor */
```

A fenti példával teljesen ekvivalens az alábbi:

```
for (i = 0, seged = vekt; i < n; i++, seged++)
{
   *seged = 0.0;
} /* endfor */
```

A különbség a két példa közt mindössze annyi, hogy a **seged** pointer a **for** ciklus inicializáló részében **vekt**-ből megkapja a valós tömbünk báziscímét, a ciklusmagban mindig a **seged** által aktuálisan megcímzett memóriaterületre írjuk be a nullát, és **seged**-et a **for** ciklus léptető utasítás-részében inkrementáljuk. Az **i** ciklusváltozó ebben a megoldásban csak számlálóként szerepel.

Tömbök dinamikus kezelésére egy másik példaként írjunk olyan függvényt, amelyik visszaadja a paraméterként kapott tömb maximális elemét:

```
double maximum(tomb, m)
float    tomb[ ];
register m;
{
   float    max;
   register i;

   max = tomb[0];
   for (i = 1; i < m; i++)
```

```
    {
        if (tomb[i] > max)
        {
            max = tomb[i];
        } /* endif */
    } /* endfor */
    return max;
} /* end maximum() */
```

Figyeljük meg a `tomb` paraméter deklarációját, ez egyenértékű a `float *tomb;` alakkal, de érzékelteti, hogy a pointert tömbként kívánjuk használni. A függvény meghívása például a következő formában történhet:

```
m = maximum(vekt, n);
```

Mivel a pointerműveletek gyorsabbak az indexeléseknél – ez utóbbiak gépi szinten szorzást is igényelnek – a fenti függvényt a gyakorlatban a következőképpen írnánk meg:

```
double maximum(tomb, m)
register float *tomb;
int             m;
{
    float  max;
    register float *veg = tomb + m;

    max = *tomb;
    while (++tomb < veg)
    {
        if (*tomb > max)
        {
            max = *tomb;
        } /* endif */
    } /* endw */
    return max;
} /* end maximum() */
```

2.9.7 Általános mutatók

A fentiekben azt láttuk, hogy a mutatóknál a mutatott típus ismerete milyen fontos, hiszen az aritmetikának is csak így van értelme (például `p++` esetén a `p` értéke, mint fizikai byte-cím annyival nő meg, mint a mutatott adat mérete byte-ban). Ennek ellenére gyakran van szükségünk "általános

pointer típusra". Gondoljunk a `malloc` függvényre: ennek értéke egy olyan pointer, aminek a típusát nem ismerjük, de nem is érdekes, úgyis átalakítjuk explicit konverzióval, mielőtt bármilyen műveletet végeznénk vele. De a `malloc`-ot is deklarálni kell valaminek! Mi legyen az? A hagyományos C implementációknál az ilyen általános pointereket `char*` típusúnak deklarálták, ami miatt nem lehetett róluk egyből látni, hogy semmi közük sincs a karakterekhez. A korszerű fordítók erre a célra bevezettek egy új típust: `void*`. Ezekkel a pointerekkel csak a következő műveletek végezhetők: tetszőleges mutatóértékkel feltölthetők, értékük típusdeklarációval tetszőleges pointerré átalakítható, vizsgálható, hogy tartalmuk `NULL`-e, vagy sem. Minden egyéb pointerművelet (például indirekció) tilos.

2.9.8 Függvényekre mutató pointerek

2.10 Bevitel és kivitel

Bár maga a C nyelv definiciója az eddig leírtakon kívül lényegében nem tartalmaz egyebet, maga a szabvány definiál ún. standard könyvtárakat, illetve standrd könyvtári függvényeket. Ezek olyan előredefiniált makrók, illetve lefordított és tárgykódú könyvtárakba szervezett függvények, amelyek majdnem minden C, vagy C++ rendszerben megtalálhatók, és hívási módjuk, illetve deklarációjuk az ANSI szabványnak megfelel. A következőkben a be- és kimenetet megvalósító szabványos függvények alapelveit, illtve a függvények közül néhányat tekintünk át. (Maguk a függvénydeklarációk, illetve a kapcsolódó makródefiniciók szabványos include file-okban találhatók.)

Ezek a függvények – a UNIX filozófiáját átvéve – eszközfüggetlenek, tehát egy program mindig úgy tekintheti az adatátvitel, hogy file-ból olvas és file-ba ír, jóllehet egy konkrét alkalmazásnál a program bemenete valójában többnyire a terminál billentyűzete, kimenete pedig a tarminál képernyője, vagy egy nyomtató. A másik általános alapelv, hogy az adatátvitel egysége a byte, és a file-ok hossza is byte-ban meghatározott. A C nyelvben a byte, mint fogalom nem létezik, ezért itt `char` használandó. A be- és kivitel tárgyalását két fő csoportba oszthatjuk: az alacsony szintű (*low-level*) és a folyam jellegű (*stream*) adatátvitelre.

A C-ben a file-ok kezelésének az MS-DOS operációs rendszer alatt kétféle módja van. Ennek az az oka, hogy a C nyelvben definiált `'\n'` újsor karakternek az operációs rendszerben a CR-LF karakterpáros felel meg. Azért, hogy szövegfile-okat portábilis módon olvashassunk, a megfelelő C rutinok elvégzik a szükséges konverziókat, eltüntetve számunkra ezt a rendszersajátosságot. Ha viszont bináris adatokat kezelünk, akkor a fenti kon-

verzió összekeveri és elrontja azokat. Ezért minden file-hoz hozzá kell rendelni egy kezelési módot megnyitáskor, ha ezt nem tesszük meg, az alapértelmezés általában a szöveges, konverziós üzemmód, de ez beállítható binárisra is. Az előre megnyitott standard állományok kezelési módja szöveges. A file kezelési módja független attól, hogy alacsony szintű, vagy folyam jellegű függvényekkel manipuláljuk.

2.10.1 Alacsony szintű I/O

Az alacsony szintű be/ki műveletek a file-leíró (*file descriptor, file handle*) segítségével hivatkozhatnak a használni kívánt állományokra. A file-leíró egy kis egész szám, amely kétféle lehet: az operációs rendszertől induláskor kapott, vagy egy könyvtári függvény használatával nyert. Az induláskor kapott (*predefined*) file-leírók a 0, 1 és 2. A 0 az ún. standard bemenet (általában a terminál, de DOS szinten átirányítható, lásd <,|), az 1-es a standard kimenet (általában a képernyő, de szintén átirányítható), a 2-es pedig a standard hibakimenet, amely mindig a képernyőre kerül. Ezek portábilisak minden implementációnál. A BORLAND C++-ban rendelkezésünkre áll a fentieken kívül a 3-as leíró, amely a COM1 soros vonalat, és a 4-es, amelyik a PRN printert kezeli. A 0 és 3 használható bevitelre, a többi – és a 3-as is – kivitelre. Tetszőleges file megnyitható az `open` könyvtári függvény segítségével – leírását lásd a függvények ismertetésénél – és a visszaadott érték sikeres megnyitás esetén a file-leíró (egyébként -1). A file-leíró birtokában tetszőleges helyről tetszőleges számú byte-ot kiírhatunk az adott állományba az aktuális pozíciótól kezdve a `write` függvény segítségével, illetve az aktuális file-pozíciótól olvashatunk adott területre tetszőleges számú byte-ot (legfeljebb a file végéig) a `read` használatával. A `read` és a `write` az átvitelt minden konverzió nélkül végzi. Vigyünk ki például az `fd` leíróval kezelt file-ba egy lebegőpontos tömböt:

```
float vekt[20];
...
n = write(fd, (char *)vekt, sizeof(vekt));
```

A forrásterületet karaktermutatóként várja a függvény, ezért alkalmaztunk típuskonverziót, a kiviendő byte-ok számának meghatározását a fordítóra bíztuk. Ha a visszaadott érték ezzel nem egyezik meg, akkor file-kezelési hiba történt. Az adatok visszaolvasása analóg módon történik. Hasznos lehet még az `lseek` függvény is, ezzel a file-ba írás illetve file-ból olvasás aktuális pozícióját állíthatjuk a kívánt helyre (a file-t mint egy lassú elérésű tömböt használhatjuk). Ha egy állományt le akarunk zárni, a `close` függvényt hívjuk meg.

2.10.2 Folyam jellegű I/O

A folyam jellegű állománykezelésnél az alapvető művelet egy karakter beolvasása és kiírása. A folyamok azonosítása file-mutatókkal történik, amelyek szerepe hasonló az alacsony szintű kezelésnél használt file-leíróéhoz. A `FILE` egy, a rendszer által az `stdio.h` include file-ban definiált struktúra, és a file-mutató egy ilyen struktúrát megcímző pointer. A file-mutatók között szintén vannak előredefiniáltak, ezek az `stdin`, `stdout`, `stderr`, és ugyanaz a jelentésük, mint a 0-ás, 1-es és 2-es file-leíróknak. (A 3-as és 4-es leíróknak az `stdaux` és `stdprn` előre definiált file-mutatók felelnek meg.) Egyéb file-mutatók a `FILE* fopen(const char* name, const char* mode)` deklarációjú könyvtári függvény hívásával nyerhetők egy adott állomány megnyitása kapcsán. (A `name` sztring a megnyitandó állomány neve, `mode` pedig az állomány kezelési módját – írás, olvasás, hozzáfűzés – határozza meg.) A file-megnyitási hibákat a visszaadott `NULL` érték jelenti. Egy karakter beolvasását a `getc()` makró végzi:

```
c = getc(fp);
```

Itt `fp` egy korábbi `fopen` hívás által visszaadott értéket tartalmazó, `FILE *fp;` definíciójú változó. A `getchar()` makró a `getc(stdin)` makróhívásnak egy rövidített formája. Ha a visszaadott érték az `stdio.h`-ban definiált `EOF` szimbólum, akkor az adott file végére értünk. A kivitel a `putc(c, fp)` makróval történik, ahol `c` a kiirandó karakter. A `putchar(c)` hasonlóan a `putc(c, stdout)` rövidítésére szolgál. Ezekben makrókban a paraméter csak egyszer kerül kiértékelésre, így nem kell tartanunk kellemetlen mellékhatásoktól, ha bonyolultabb kifejezéseket használunk aktuális paraméterként. E makrók használatára vonatkozó példákat a `pelda.c` programban találhatunk.

Formátumozott outputot a `int printf(const char* format, ...)`, illetve az `int fprintf(FILE* outfile, const char* format, ...)` deklarációjú függvényekkel állíthatunk elő. Az előbbi a standard outputra (`stdio`), míg az utóbbi az `outfile` file-pointer által meghatározott, írható file-ba nyomtat. A nyomtatási képet a `format` sztring határozza meg. A függvények visszatérési értékül a formátum specifikáció szerint kinyomtatandó paraméterek számát adják.

A `format` sztring kétféle karaktert tartalmazhat. Egyrészt normál karaktereket, amelyek minden további nélkül a kimeneti állományba másolódnak, másrészt konverzió specifikációkat. A konverzió specifikációk hatására a `format` sztringet követő további argumentumok kerülnek rendre kinyomtatásra az egymást követő specifikációk alapján. Minden specifikáció egy százalékjel (`%`) karakterrel kezdődik, és egy vezérlő karakterrel zárul. A `%` és a vezérlő karakter között még az alábbi karakterek állhatnak (a felsorolás

szerinti sorrendben):

- *Egykarakteres jelző flag.*
  Elhagyása esetén vezető nullákkal, vagy szóköz karakterekkel kitöltve, az adott mezőben jobbra igazítva keletkezik az output. Ha megadunk valamilyen flag-et, akkor az a következő lehet: Egy - (minusz jel), ami azt jelenti, hogy a konvertált argumentumot balra igazítva kell kinyomtatni, vagy egy + (plusz jel), ami egészek esetében előírja, hogy az előjel mindig ki legyen nyomtatva. Ha flag-ként a # (*hash-mark*) karaktert adjuk meg, akkor egy ún. alternatív nyomtatási képet kapunk. Az `o` vezérlő karakter esetén minden nem nulla szám vezető `0`-val, `x`, vagy `X` vezérlő karakterek esetén minden esetben `0x`, illetve `0X` előtaggal lesz kinyomtatva. Az `e`, `E` és `f` vezérlő karakterek esetén mindig kapunk tizedes pontot, `g` és `G` esetén szintén, de a vezető `0`-k elmaradnak.

- *Mezőszélesség*
  Ez egy egész szám, amelyik a minimális mezőszélességet adja meg. A konvertált argumentum legalább ilyen széles mezőn lesz kinyomtatva, szükség esetén az üres helyek szóköz karakterrel lesznek kitöltve. Ha `0`-val kezdődik a mezőszélesség, akkor a kinyomtatott számok vezető `0`-kat kapnak. Ha a * karaktert adjuk meg mezőszélességként, akkor a tényleges mezőszélesség értéket az argumentumlistában soron következő adat határozza meg. (Ez az érték nem kerül konvertálásra.)

- Egy pont, ami a mezőszélességtől elválasztja a pontosságot előíró értéket.

- *Pontosság*
  Ez egy egész szám, amelyik megadja, hogy egy sztringből maximálisan hány karakter kerüljön kinyomtatásra, vagy megadja egy lebegőpontos szám tizedes pontja után nyomtatandó értékes jegyek számát, vagy egy egész szám minimálisan nyomtatandó jegyeinek a számát írja elő. Ha a pontosság a specifikációból kimarad, akkor a konvertálás során az alapértelmezés szerinti érték (6) lesz figyelembe véve. Ha `0`-t írunk elő, akkor a `d`, `i`, `o`, `u`, `x`, illetve `X` vezérlő karakterek esetén az alapértelmezés szerinti számú jegy lesz kinyomtatva, az `e`, `E` és `f` vezérlő karakterek esetén elmarad a tizedes pont. A * karakter megadása azt jelenti, hogy a tényleges pontosság értéket az argumentumlista soron következő eleme tartalmazza.

- *Módosító előtag*
  A módosító előtag karakter közvetlenül a vezérlő karakter előtt áll.

Lehetséges értékei a következők: Egy `h`, ha egy egész számot `short`-ként, vagy egy `l` (el betű), ha `long`-ként kell nyomtatni. A BORLAND C++-ban pointerek nyomtatása esetén alkalmazhatjuk még az `N`, illetve az `F` előtagokat is. Az előbbi hatására `near`, míg az utóbbi hatására `far` pointer-formátumú lesz a konvertált érték.

Mint említettük, a mezőszélesség vagy a pontosság helyett megadhatjuk a * karaktert is. Ebben az esetben a megfelelő értéket az eddig feldolgozott argumentumokat követő első argumentum – ami egy `int` típusú kifejezés kell legyen – fogja meghatározni. Például a

```
printf("%.*s",max,str);
```

utasítás hatására az `str` sztringből a `max` változóban lévő érték által meghatározott számú karakter kerül a standard kimenetre.

A formátum sztringben használható vezérlő karaktereket a 2.6. táblázatban foglatuk össze.

Figyelem! A `printf` függvény az első paramétere alapján határozza meg, hogy még hány, és milyen típusú további paramétere van. Mindenféle "szemét" kerülhet az outputra, ha a formátum sztringet nem a specifikációnak megfelelő számú, vagy típusú paraméter követi. A `printf`-ről elmondottak természetesen a `printf`-függvénycsalád többi tagjára (`fprintf`, `sprintf`) is vonatkoznak. (Az `sprintf` függvény ugyanazt végzi, mint az `fprintf`, de file helyett egy karaktertömbbe "nyomtat").

Az `int scanf(const char * format, ...)` deklarációjú függvény a `printf` függvénynek megfelelő beolvasó rutin. Ez a függvény az `stdin` állományról olvas be adatokat, és az esetek többségében a `printf` konverziók fordítottját hajtja végre.

A `scanf` karaktereket olvas a standard inputról, a `format` formátum-specifikáló sztring szerint értelmezi és konvertálja azokat, és a további argumentumai – amelyek a formátum specifikációnak megfelelő típusú tárolási egységekre mutató pointerek – által meghatározott memória helyekre írja be a beolvasott értékeket.

A `scanf` befejezi az olvasást, ha a specifikációnak megfelelő számú adatot már beolvasott, vagy ha az input valamilyen oknál fogva nem felel meg formátum-specifikációnak. Visszatérési értékül a sikeresen beolvasott adatok számát adja meg a `scanf`. Ez jól használható arra, hogy ellenőrizzük, tényleg annyi adatot olvasott-e be a programunk, mint ahányat kellett. Ha file vége után olvasnánk a `scanf`-fel, akkor visszatérési értékül az `EOF`-ot (*end of file*) kapjuk. Az `EOF` szimbólum az `stdio.h` include file-ban van definiálva. Ha visszatérési értékül 0-t kapunk, ez azt jelzi, hogy a soron következő beolvasandó karakter nem felel meg a formátum-specifikációnak.

KARAKTER	ARGUMENTUM TÍPUS	NYOMTATÁSI KÉP
`d, i`	`int`	előjeles decimális egész
`o`	`int`	előjel nélküli oktális egész
`x, X`	`int`	hexadecimális egész a vezető `0x`, vagy `0X` előtag nélkül
`u`	`int`	előjel nélküli decimális egész
`c`	`int`	`unsigned char`-rá való konvertálás után a megfelelő karakter
`s`	`char*`	sztring karaktereit nyomtatja `EOS`-ig, vagy az adott mezőszélesség határáig.
`f`	`double`	[-]*m.dddddd*, ahol a *d*-k száma az előírt pontosságtól függ. A pontosság alapértelmezésben 6, a 0 pontossággal a tizedes pont nyomtatása elnyomható
`e, E`	`double`	[-]*m.dddddd* *e*± *xx*, vagy [-]*m.dddddd* *E*± *xx*, ahol a *d*-k száma az előírt pontosságtól függ.
`g, G`	`double`	ugyanaz, mint `%e`, vagy `%E`, ha az exponens kisebb, mint -4, vagy nagyobb-egyenlő, mint a pontosság egyébként ugyanaz, mint `%f`. Nincsenek vezető 0-k, vagy szóközök.
`p`	`void*`	pointer-érték; implementáció függő, a BORLAND C++-ban tármodelltől is függ.
`%`	—	nincs konverzió, maga a `%` karakter nyomtatódik

2.6. táblázat: A `printf` függvénycsalád formátum-specifikáló sztringjeinek vezérlő karakterei

A `format` sztring az alábbi karaktereket tartalmazhatja:

- Tetszőleges *whitespace* karakter, amelyeket a a `scanf` figyelmen kívül hagy.
- Normál karakterek a `%` kivételével. Ezek meg kell hogy egyezzenek az input folyam első nem *whitespace* karakterével.
- Konverzió specifikációk. Ezek a `%` karakterrel kezdődnek és opcionálisan tartalmazhatják a `*` karaktert (melynek hatására a beolvasott adatot átugorja a `scanf`), egy mezőszélességet megadó számot, a beolvasott adat tárolási méretére (például `short` vagy `long` egész, `float` vagy `double` lebegőpontos szám) utaló `h`, `l` (kis ell), vagy `L` karaktereket és vezérlő karaktereket. A `scanf` formátum-specifikáló sztringjének vezérlő karaktereit a 2.7. táblázatban foglatuk össze. A BORLAND C++ lebegőpontos számok beolvasása esetén az `L` adatszélességre utaló karaktert a `long double` típus számára tartja fenn. A BORLAND C++-ban megadható még az `N` és az `F` karakter is, amellyel a beolvasandó pointerek `near`, illetve `far` értelmezésére utalhatunk.

A `d`, `i`, `o` és `x` konverziót vezérlő karakterek előtt állhat a `h` előtag, amely azt jelzi, hogy `int*` helyett `short*` típusú pointer áll az argumentum-lista megfelelő helyén, illetve az `l`, vagy `L` előtaggal azt jelezhetjük, hogy `int*` helyett `long`-ra mutatató pointer található az argumentum-listában. Ez utóbbi esethez hasonlóan az `e`, `f` és `g` vezérlő karakterek előtt álló `l`, vagy `L` előtag arra utal, hogy `float` helyett `double` adattal van dolgunk. A BORLAND C++-ban az `L` előtag a `long double` típusú adatok specifikálására szolgál.

Ahogy a formátumozott outputtal kapcsolatban függvénycsaládról beszéltünk, ugyanígy létezik a beolvasásra is a `scanf` függvénycsalád. Ha `stdin` helyett egy valamilyen más állományra vonatkozó adatfolyamból szeretnénk olvasni, akkor az `fscanf` függvényt használjuk. Sztringből való "olvasásra" az `sscanf` függvény szolgál.

Mind a `printf`, mind a `scanf` függvénycsalád esetében a `n` vezérlő karakter hatására a már konvertált karakterek száma a specifikációhoz tartozó `int*` típusú az argumentum által megcímzett memóriahelyre kerül; nem történik semmiféle konverzió, és a konvertált adatok számlálója sem inkrementálódik.

KARAKTER	INPUT ADAT	ARGUMENTUM TÍPUS
`d`	decimális egész	`int*`
`i`	egész szám akár oktális (vezető `0`), akár hexadecimális (vezető `0x`, vagy `0X`) formában	`int*`
`o`	oktális egész a vezető `0` előtag nélkül	`int*`
`u`	előjel nélküli decimális egész	`unsigned int*`
`x`	hexadecimális egész, akár a `0x`, vagy `0X` előtaggal, akár anélkül	`int*`
`c`	karakter(ek). A soron következő karakter (alapértelmezésben csak 1) kerül a megfelelő memóriaterületre. A *whitespace* karakterek is beolvasásra kerülnek. következő nem *whitespace* karakter olvasásához a `%1s` specifikációt használjuk.	`char*`
`s`	sztring beolvasása. A specifikációhoz tartozó pointer egy olyan tömbre kell mutasson, amely elegendően nagy a beolvasandó karakterek és a sztring végét jelző `EOS` tárolására.	`char*`
`f`, `e`, `g`	E három vezérlő karakter szolgál lebegőpontos számok beolvasására. Az előjel, a tizedes pont és az exponens opcionális. Ha a specifikációban az `l`, vagy az `L` előtag is szerepel (például `%lf`), akkor az argumentum típusa:	`float*` `double*`
`p`	a `printf` által produkált formátumú pointer értéket olvassa be	`void**`
`%`	nincs hatása	–

2.7. táblázat: A `scanf` függvénycsalád formátum-specifikáló sztringjeinek vezérlő karakterei

2.11 A main függvény

Minden C program kötelezően tartalmaz egy **main** nevű függvényt. Definiálásának helye lényegtelen, akármelyik forrásfile 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, ha nem adunk meg típust a definicióban, 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** batch funkcióval.

Amennyiben programunkat nem a **COMMAND.COM** indította el, hanem egy másik felhasználói program hívta (lásd a **spawn** és **exec** függvénycsaládot a **process.h** include file-ban), akkor az megkapja a visszaadott státuszkódot, és felhasználhatja annak eldöntésére, hogy programunk sikeresen futott-e 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, file-kezelési hiba, *CTRL BREAK*, stb.). Minden komolyabb programnak illik visszaadni státuszkódot – erre legtöbbször az **exit** függvényt használjuk (melyet szintén a **process.h** file-ban 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 3 paramétert kap, de ezek közül csak annyit kell átvennie, amennyire ténylegesen szüksége van. Elhagyni azonban csak hátulról előre haladva lehet a paraméterlistáról. A **main** komplett 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 (*environment variables*) és értéküket tartalmazó sztringek mutatótömbje. Ha programunk neve **PROG**, akkor a

```
SET KV=ALMA
PROG alfa beta GAMMA
```

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

- **argc** értéke 4,

- `argv[0]` a `PROG` teljes elérési útjára mutat (bár megjegyzendő, hogy a DOS 3.0-ás, vagy korábbi verziói esetén `argv[0]` üres sztringre 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 (például `PATH`, `COMSPEC`, `PROMPT`). Az `env` tömb végét szintén a `NULL` értékű elem jelzi.

2.12 További magyarázatok

2.12.1 A balérték és jobbérték fogalma

Egy tárterület-foglaló tárolási egységre vonatkozó kifejezést *balértéknek* (angolul *lvalue*) nevezzünk. Ez nyilvánvalóan konstansokat, változókat és dinamikusan lefoglalt memóriaterületeket jelent. (A konstansok olyan tárterület-foglaló egységek, amelyeknek nincs azonosítójuk.)

Szorosabb értelemben véve a balérték egy olyan tárolási egységre vonatkozó kifejezés, amelynek érték adható. Nyilvánvalóan balérték egy megfelelő típusú és tárolási osztályú változó azonosítója.

Vannak olyan operátorok, amelyek eredményül balértéket szolgáltatnak. Például ha `E` egy tárterület-foglaló tárolási egységre mutató pointer típusú kifejezés, akkor `*E` egy olyan változó, amire `E` mutat.

A *balérték* elnevezés az értékadó operátorral kapcsolatos: az `E1 = E2` típusú értékadó műveletek bal oldalán álló `E1` operandus balérték kell legyen. Röviden úgy is fogalmazhatunk, hogy minden olyan kifejezés, amelynek érték adható, balérték. Minden olyan kifejezést, amit értékül adhatunk, *jobbértéknek* (angolul *rvalue*-nak) nevezünk. A fentiek alapján persze az is nyilvánvaló, hogy nem minden balérték szerepelhet értékadás baloldalán (lásd a konstansokat). Tekintsük az alábbi definiciókat:

```
char buffer[256];
char *chptr, ch1;
const char xchar = 'x';
```

```
char *get_buff_pos(int i);
{
  if (i < 0 || i > 255)
  {
     return &ch1;
  }
  else
  {
     return buffer+i;
  }
}
```

A fenti definiciók szerint helyes értékadások lehetnek az alábbiak:

```
chptr = get_buff_pos(12); /* &buffer[12] lesz */
ch1 = xchar;
*get_buff_pos(1) = 'b';
```

A fenti értékadásokban szereplő balérték kifejezések rendre `chptr`, `ch1`, `*get_buff_pos(1)`. Az első két példa triviális: mindkét esetben egyszerű változóról van szó, melyeknek minden további nélkül érték adható. A harmadik példa azt mutatja be, hogy használhatjuk ki azt, hogy egyes operátorok balértéket képeznek. A függvények visszatérési értéke a C-ben nem használható fel balértékként (ezen változtat majd a C++-ban az ún. referencia típusú visszatérési érték – lásd a 3.1.2-es részt), de az indirekció oerátorral, a *-gal a `get_buff_pos` függvény karakterre mutató pointer típusú visszatérési értékéből azonnal egy tényleges változót – a `buffer` tömb 1. indexű elemét – állítjuk elő, amely már természetesen felhasználható balértékként.

Felmerülhet az a kérdés, hogy `xchar` miért nem szerepelt a fenti értékadó utasítások közt baloldalon. Nos azért, mert ezt a "változó"-t a `const` módosító szóval definiáltuk, és mint már említettük, az így definiált tárolási egységeket konstansnak tekinti a fordító, azaz értékük módosítását nem teszi lehetővé, bár maga az `xchar` kifejezés balérték.

2.12.2 Még egyszer a deklarációkról és a típusokról

Amint arra a 2.9.1-es részben utaltunk, az alaptípusokból (és persze a felhasználó által definiált típusokból is) a típusmódosító operátorok felhasználásával (lásd a 2.4-es táblázatot) újabb típusokat származtathatunk.

A származtatott típusok jelölésével és precedenciájával kapcsolatos ismereteket is áttekintettük. Ezek megértését talán nehezíti, hogy a * típusmódosító operátor prefix operátor, míg a tömbtípust képző [ ] operátor

és a függvénytípust képző ( ) operátor postfix operátor. Ennek következtében sokszor zárojelezni kell, hogy egy adott definició az elképzeléseink szerinti típust eredményezze. Bonyolultabb típusok esetén célszerű a `typedef` használata, de egyszerűbb esetekben ez sokszor elmarad. Gyakori, hogy egy származtatott típusú tárolási egysége deklarációja során történik a típusdefiniálás is. Erre tipikus példa a pointer változók deklarálása, mint például `int* ip;` vagy egyszerre több típusmódosító operátort is alkalmazva, amint azt az alábbi példák szemléltetik:

```
int* v[10];
int (*p)[10];
```

A fenti deklarációk szerint `v` egy 10 elemű pointer-tömb, `p` pedig egy egy 10 elemű vektorra mutató pointer. (Mivel a `[ ]` precedenciája magasabb, mint a `*` típusmódosító operátoré, a `p` ponter deklarációjakor zárójeleznünk kellett.)

Eléggé fáradtságos dolog lenne, ha egy program minden egyes azonosítóját egyenként kéne deklrálni. Ez különösen igaz abban az esetben, ha az egyes tárolási egységek azonos típusúak. Amint azt a 2.4.4-es alfejezetben láttuk, lehetőségünk van arra, hogy egy deklarációban azonosítólistát alkalmazzunk: a `int x, y;` deklaráció egyenértékű az `int x;  int y;` deklarációkkal.

Fontos azonban megjegyezni, hogy amikor a fenti példához hasonló származtatott típusú változókat deklarálunk, akkor a típusmódosító operátorok csak egyetlen listaelemre vonatkoznak. Tehát például az

```
int* p, y;
```

deklaráció az

```
int* p;
int  y;
```

deklarációval egyenértékű, azaz `y` típusa `int` lesz, nem pedig egészre mutató pointer. Az a véleményünk, hogy minden összetettebb típust definiáljuk önálló típusként a `typedef` kulcsszó segítségével, kerüljük az adatdeklaráció során a típusszármaztatást.

2.12.3 Függvények változó számú paraméterrel

A 2.10.2-es részben ismertetett `printf`, illetve `scanf` függvények (függvénycsaládok) tipikus példái a változó paraméterlistájú függvényeknek. A továbbiakban a `printf` függvény egy minimál implementációján keresztül

mutatjuk be, hogyan készíthetünk portábilis, változó hosszúságú paraméterlistával rendelkező C függvényeket. Függvényünket a következőképpen deklaráljuk:

```
void minprintf(const char *fmt, ...);
```

Azért **void** típusú a függvény, mert az egyszerűség kedvéért ebben a példában a konvertált adatok számával nem foglakozunk, így azt visszatérési értékül sem tudjuk szolgáltatni. A deklarációban a ... csak a formális paraméterlista utolsó elemeként szerepelhet, és legalább egy megnevezett argumentumot kell deklarálnunk ahhoz, hogy a változó hosszúságú paraméterlistát majd kezelni tudjuk.

A szabványos **stdarg.h** include file tartalmazza azokat a makrókat, amelyek a változó hosszúságú paraméterlisták kezeléséhez szükségesek. Az egyes makrók megvalósítása gépről gépre változhat, de a makrók egységes felületet teremtenek a pobléma kezeléséhez.

A **va_list** típus szolgál arra, hogy a soronkövetkező függvényargumentumra vonatkozó információ tárolására való változót – egy argumentum-pointert – deklarálhassunk. A **va_start( )** makróval inicializálhatunk egy ilyen változót. Az inicializálás eredményeképpen az argumentum-pointer az első azonosító nélküli argumentumra fog mutatni. Mint említettük, legalább egy azonosítóval rendelkező elemet kell hogy tartalmazzon a formális paraméterlista. Ezt használja fel a **va_start( )** makró az argumentum pointer inicializálásához. A **va_arg( )** makróval léphetünk tovább a következő azonosító nélküli argumentumra. Ez a makró a soron következő aktuális paramétert szolgáltatja értékül. Ennek a makrónak két paramétere van. Az első az argumentum-pointer, a második pedig egy típusazonosító, amely megszabja, hogy milyen legyen a visszatérési érték típusa, és hogy az argumentum-pointert milyen mértékben (hány byte-tal) kell továbbléptetni. A **va_end( )** makró szolgál arra, hogy "rendet rakjon" egy változó hosszúságú paraméterlistával rendelkező függvényből való visszatérés előtt. Ezt a makrót mindig meg kell hívni egy ilyen függvényből való kilépés alkalmával.

Ezek után nézzük meg az egyszerüsített **printf** függvény listáját!

```
#include <stdarg.h>
/*********************************************************/
 void minprintf(const char *fmt, ...)
/*                                                       */
/* Valtozo hosszusagu parameterlista kezelesenek bemutatasa */
/* a minimalis printf funkciok kapcsan. Mezoszelesseg, stb. */
/* kimarad, konverzio es nyomtatas az eredeti printf-fel.   */
/*********************************************************/
{ va_list ap;          /* argumentum pointer                */
  char    *p,          /* fmt-n szalad majd vegig           */
          *sval;       /* ->sztring tipusu parameter erteke */
  int     ival;        /* int tipusu parameter erteke       */
  double  dval;        /* double tipusu parameter erteke    */

  va_start(ap, fmt); /* ap az fmt utani elso arg.-ra mutat  */
  for (p = fmt, *p, p++)              /* p-t EOS-ig lepteti  */
  { if (*p != '%') { putchar(*p); } /* Csak masolas          */
    else              /* specifikacio feldolgozasa:        */
    { switch(*++p)
      {
        case 'd': ival = va_arg(ap, int);
                  printf("%d", ival);
                  break;

        case 'f': dval = va_arg(ap, double);
                  printf("%f", dval);
                  break;

        case 's': for(sval = va_arg(ap,char*); *sval; sval++)
                  {
                    putchar(*sval);
                  }
                  break;

        default:  putchar(*p);
                  break;
      }
    }
  }
  va_end(ap); /* Takaritas a fuggveny vegen                */
}
/*********************************************************/
```

2.13 Összetett mintapélda

Jelen példánk egy igen flexibilis menükezelő rendszer vázát tartalmazza. Az itt felhasznált megoldások sokat segíthetnek a típusdefiniciókkal, struktúrákkal, pointerekkel, függvény-pointerekkel kapcsolatban leírtak megértésében.

E példa fő célja a portábilis programozási stílus bemutatása, másrészt igyekszünk rávilágítani arra, hogy egy célszerűen megtervezett adat- és szubrutinstruktúra mennyire áttekinthetővé és könnyen módosíthatóvá teszi a felhasználói programjainkat. Felhívjuk az olvasó figyelmét, hogy ezzel a példprogrammal nem azt akarjuk sugallni, hogy ez az igazi menükezelő, illetve felhasználói felület. Léteznek olyan objektum-orientált könyvtárak, amelyek az itt leírtaknál sokkal fejletteb felhasználói felületet valósítanak meg – természetesen használatukhoz ismerni kell a C++-t, illetve ha Windows alkalmazói programot készítünk, a programvezérlésről és a menükről alkotott képünket mindenképpen át kell alakítanunk.

2.13.1 A tervezés egyes fázisai

Minden programfejlesztési munka során előjön az a feladat, hogy az adott program számára egy felhasználói felületet (*user interface*-t) kell írni. Ez a felület az esetek legnagyobb részében valamilyen menürendszert jelent. Egy menürendszert úgy célszerű kialakítani, hogy az általa nyújtott szolgáltatások az egész felhasználói programban igénybevehetők legyenek, és a felhasználói program többi részében lehetőség szerint ne kelljen a képernyő- és billentyűzetkezeléssel foglalkozni. Az egész felhasználói program, illetve a hozzákapcsolódó menürendszer tervezésekor egy másik fontos szempont az, hogy a program belső vezérlési szerkezetei tükrözzék azt a vezérlési szerkezetet, amit a felhasználó észlel a program használata során. Másképpen ezt úgy fogalmazhatjuk meg, hogy ne az egyes programfunkciók aktivizáljanak kisebbnagyobb menürutinokat, hanem egy átfogó, hierarchikus menürendszer gondoskodjon arról, hogy mindig a felhasználó kívánsága szerinti programfunkciók legyenek aktivizálva.

Portabilitási megfontolások

Ha fáradtságos munkával megtervezünk és létrehozunk egy, a fenti kívánalmaknak megfelelő felhasználói felületet, célszerű azt úgy programozni, hogy ne csak IBM-PC kompatibilis számítógépeken, a DOS operációs rendszer alatt, BORLAND C++ fordítóval lefordítva fusson, hanem jól körülhatárolt módosítások után bármely, C fordítóval rendelkező géptípuson, bármely operációs rendszeren (pl. VT100-as terminálokkal rendelkező VAX

gépeken Ultrix operációs rendszerben) is használhassuk a megírt rutinjaink többségét.

Ennek érdekében célszerű a megírandó menürendszert 3 részre osztani. Az első rész tartalmazza a legmagasabb szintű függvényeket, amelyek változtatás nélkül portábilisak. A második, közbenső szint tartalmazza azokat a függvényeket, amelyeknek törzsét az aktuális C fordító és operációs rendszer rendszerfüggvényei, illetve az aktuális számítógép-konfiguráció képernyője szerint módosítani kell. A harmadik, legalacsonyabb szinten célszerű elhelyezni a teljesen hardver-specifikus függvényeket. Ilyenek lehetnek például az IBM PC BIOS rutinhívások.

Jelen példánkban csak a legmagasabb szintű részeit mutatjuk be menükezelő rendszerünknek. A második, és harmadik csoprtba tartozó függvények közül csak a közvetlenül felhasznált függvények deklarációit közöljük rövid magyarázatokkal.

A látvány (*look-and-feel*) megtervezése

Alapvetően a BORLAND C++ integrált fejlesztői környezetének menükoncepcióját igyekszünk megvalósítani a *hot key*-k kivételével. Igyekszünk egy egyszerű help-rendszert is megvalósítani, de nem célunk a BORLAND C++ környezetfüggő rendszerének a lemásolása.

A menürendszert úgy látja a felhasználó, hogy több alfanumerikus ablak van a képernyőn. A BORLAND C++ erre ténylegesen is lehetőséget nyújtana, de a hordozhatóság miatt ezt nem használjuk ki. A menükezelő rendszerben az összes karakternyomtató utasítás az egész képernyőre vonatkozik, mi magunk figyelünk arra, hogy csak a képernyő bekeretezett részén történjen nyomtatás. A képernyőn mi magunk hozunk létre keretezett részeket, dobozokat az IBM PC kiterjesztett karakterkészletével. A felhasznált 'jobb felső sarok', 'bal felső sarok', 'függőleges vonal', stb. karakterek egyes számítógép terminálokon is léteznek, "természetsen" más kódokkal, így célszerűen ezeket például a `#define` direktívával szimbólumokhoz rendeljük.

Minden menü egy ilyen dobozba kerül, az egyes almenük dobozai a szülő menü dobozától egy kicsit lejebb kerülnek a képernyőre. Készítünk egy főmenü keretet is. Ennek a legfelső sora lesz a főmenű, azaz az egymástól független menüfák gyökerének a gyűjtőhelye. A főmenűből az egyes menüpontokat vagy egy dedikált billentyű leütésével, vagy a kurzor-mozgató nyilak (←, illetve → nyilak) és az *Enter* billentyű segítségével választhatjuk ki. A kiválasztás hatására a menüpont alatt megjelenik a megfelelő almenü kerete, benne az egyes almenüpontokkal. Egy almenüponthoz vagy egy közvetlenül végrehajtható programrész, vagy egy további almenü tartozik. Az almenük pontjait a ↑ ↓ kurzorvezérlő billentyűk és az *Enter*, illetve dedikált

billentyűk segítségével választhatjuk ki.

Egy almenüből az *Esc*, vagy a minden menüben szereplő e**X***it* menüponthoz rendelt *X* billentyű leütésével léphetünk ki. (Az e**X***it* menüpontot és a hozzá tartozó *X* billentyűt a portabilitás miatt definiáltuk: egyes terminálokon az *Esc* billentyű kódja terminálvezérlő karakterszekvenciák része, így e billentyű leütését vagy nem tudjuk érzékelni, vagy a terminál "megbolondul" tőle.) Egy menüpontként aktivizált programrészből, vagy egy almenüből visszatérve a hívó menü képe mindig regenerálódik, és az utoljára aktivizált menüpont marad kiválasztva, azaz egyszerűen csak az *Enter* billentyű leütésével újra aktivizálható.

Leképezés adatstruktúrákra és vezérlési szerkezetekre

Az előbb vázolt megjelenés a képernyőn, illetve kezelési mód azt sugallja, hogy szükségünk van egy, a főmenüt leíró adatstruktúrára és az azt kezelő főmenü függvényre, illetve létre kell hozni egy olyan adatstruktúrát, amellyel leírhatjuk, hogy egy almenü hol helyezkedik el a képernyőn, milyen menüpontjai vannak, azokhoz milyen funkció (milyen végrehajtandó programrész, vagy milyen további almenü) tartozik, stb.

Nyilvánvaló tehát, hogy kell egy adatstruktúra, ami az egyes menüpontokra vonatkozó információkat tartja nyilván (a menüpont neve, a hozzá tartozó help-információ, a hozzárendelt kiválasztó billentyű, kiválasztották-e, milyen feladatot lát el, esetleges paraméter). A menüpontokat menülistákba kell szerveznünk. Egy ilyen listát ki kell egészítenünk a képernyőn való megjelenésre vonatkozó információkkal (milyen hosszú a lista, hány karakter széles, a listát tartalmazó doboz hol helyezkedik el a képernyőn, stb.), és megadhatjuk azt is, hogy egy adott menü milyen hierarchia szinten helyezkedik el a menü-fán.

A *look-and-feel*-re vonatkozó meggondolásokból következik, hogy az almenüket kezelő menüfüggvényt ugyanolyan funkcióként érdemes felfogni, mint a programunk ténylegesen végrehajtandó egyes részeit. Így tehát célszerű az egyes menülistákat megszámozni, és a menükezelőnk ezen szám alapján tudja eldönteni, melyik menülistát kell megjelenítenie és kezelnie. Természetesen az is célszerű, hogy az egyes menüpontok a végrehajtandó programrészletekre vonatkozó default paramétereket tartalmaznak, és a menüpont kiválasztásakor ezen paraméterrel hívja meg a menükezelő a megfelelő részprogramot.

Ilyen meggondolások mellett egy almenü megjelenítése belső vezérlési szerkezetként úgy nyiványul meg, hogy a menükezelő függvény önmagát hívja meg úgy, hogy a rekurzív hívás alkalmával az almenü azonosítóját, mint paramétert használja.

Hogy valósítsuk meg egy adott menüponthoz tartozó függvény aktivizá-

lását? A válasz igen egyszerű: indirekt függvényhívást kell alkalmaznunk, azaz a menüpont leíró struktúrában egy függvényre mutató pointermezőt kell deklarálnunk. Az egyes menüpontok definiálásakor ezt a struktúramezőt a ténylegesen meghívandó függvény címével kell majd inicializálnunk.

A konkrét deklarációk

Most tekintsük tehát az egyes típusdeklarációkat! A menürendszerünk különböző menükből áll, a különböző menük pedig több menüpontból. Egy menüpont legfontosabb jellemzője az a függvény, amit a menüpont kiválasztásakor aktivizálni kell. Ezek a függvények igen sokfélék lehetnek, így hagyományos C-ben célszerű a függvények címeit nyilvántartani. Ehhez két lépcsőben definiáljuk a `fad` (function address) típust:

```
typedef int intfunc(int);/* int-et visszaado, 1 int-et varo *
                          * fuggvenytipus                    */

typedef intfunc *fad;    /* intfunc tipusu tarolasi egy-  *
                          * segre mutato tipus              */
```

A fent definiált `intfunc` típust felhasználhatjuk a majdan meghívandó egyes függvények előzetes deklarálására.

A végrehajtandó függvényen kívül egy menüpont fontos jellemzője az illető menüpont neve (azonosító szövege), az a nyomtatható karakter, amivel *Enter* megnyomása helyett kiválasztható a menüpont. Célszerű megengednünk, hogy a menüpont által meghívandó függvénynek egy, a menüpont leírásában tárolt paramétert is átadjunk. Ha a menürendszerünkhöz alkalmas help-rendszert is szeretnénk, célszerű az egyes menüpontokhoz rendelt help-szövegre utaló információt (például egy file-indexet) is tárolni. Ezeket az informáiókat – a menüponthoz rendelt függvény címével együtt – az alább deklarált `menuitem` (menüpont) struktúrába szerveztük:

```
typedef struct
{
   char *text;          /* A menupont azonosito szovege      */
   char key;            /* A menupontot kivalaszto betu      */
   int  helpindex;      /* A menuponthoz rendelt help-kod    */
   fad  function;       /* A menuponthoz tartozo fv-re mutat */
   int  param;          /* A '*function' fuggveny parametere */
} menuitem;
```

A figyeljük meg, hogy a fenti struktúra definicóból kimaradt a típuscímke, hiszen `typedef`-fel eleve azonosítót rendelünk hozzá – rekurzív adatdefinicóról pedig szó sincs.

Most lássuk, hogy szervezhetünk egy menüt a fenti módon deklarált **menuitem** struktúrák segítségével.

A menüpontjainkat célszerűen egy **menuitem** típusú tömbben tároljuk, amelynek méretét is tudnunk kell. A menü tartalma mellett fontos annak megjelenése is. Szükségünk lehet arra, hogy a menüt keretező doboz tetején esetleg egy menünevet, egy fejlécet (header-t) is megjelenítsünk. Fontos azt is tudnunk, hogy melyik x-y karakterpozicióba kerül a menüdoboz (annak például a bal felső sarka) a képernyőn, és az is lényeges információ, hogy hány karakterpoziciót foglal le a menüdoboz vízszintes és függőleges irányban. Azt is nyilvántarthatjuk egy menüről, hogy melyik menüpontot választottuk ki benne utoljára és fontos lehet az is, hogy az adott menü hol helyezkedik el egy hierarchikus menü-fán. Ezeket az információkat foglaltuk egybe az alábbi **menutype** struktúrában:

```
typedef struct
{
  char     *header;  /* A menu fejlecszovegere mutat          */
  int      x;        /* A menudoboz bal felso sarkanak        */
  int      y;        /* x es y koordinatai, valamint          */
  int      xs;       /* a menudoboz x es                      */
  int      ys;       /* y iranyu merete.                      */
  int      itemno;   /* A menupontok szama                    */
  menuitem *items;   /* A menupontok listajara mutat.         */
  int      hierarch; /* Ha 1, kozvetlenul a fomenu hivja      */
  int      lastitem; /* Utoljara kivalasztott pont szama      */
} menutype;
```

A **menuitem** típusból egy-egy inicializált tömböt szervezve hozhatjuk létre az egyes menük tartalmára vonatkozó adathalmazt. Egy ilyen lista kezdőcíme kerül egy **menutype** struktúra **items** mezőjébe. Egy-egy **menutype** struktúra egy komplett menü leírását tartalmazza. Ezekből a struktúrákból szintén egy tömböt szervezünk, ez lesz a **menus** tömb. E tömb első néhány eleme egy-egy menüfa gyökerét (azaz a főmenü egyes pontjaiként aktivizálandó menüket) reprezentálja, a többi elem pedig az egyes fákra felfűzött almenüket írja le. Tekintsük át tehát a teljes menürendszert definiáló adatstruktúrát:

```
/* Kulso fuggvenyek deklaracioja                              */

extern intfunc data,  r_data, w_data, statf,
              regr,  linf, barf,
              save,  load;
```

```
/* A a menukezelo fuggveny prototipus erteku deklaracioja   */

intfunc menu;        /* Elore hivatkozashoz                  */
intfunc dir, shell; /* Tovabbi fv-ek elore hivatkozshoz      */

/* Az egyes menulistak (items_0 .. items_3) es a menuk:     */

menuitem items_0[ ] =
{ /* text       key hlp func. param.    */
  "Directory", 'D', 1, dir,   0,
  "Os shell",  'O', 2, shell, 0,
  "File",      'F', 3, menu,  3,/*a 3.sz. menu almenu lesz  */
  exitxt,      'X',-1, NULL,  0 /*-1-es parameter: exit     */
};                                /* Tombmeret:              */
#define N0 sizeof(items_0)/sizeof(menuitem)

menuitem items_1[ ] =
{
  "Default",   'D', 4, data,  7,
  "Read data", 'R', 5, r_data,1,
  "List data", 'L', 6, w_data,2,
  "Statistics",'S', 7, statf, 3,
  exitxt,      'X',-1, NULL,  0
};
#define N1 sizeof(items_1)/sizeof(menuitem)

menuitem items_2[ ] =
{
  "Regression",'R', 8, regr,  4,
  "Plot",      'P', 9, linf,  5,
  "Bar",       'B',10, barf,  6,
  exitxt,      'X',-1, NULL,  0
};
#define N2 sizeof(items_2)/sizeof(menuitem)

menuitem items_3[ ] =
{
  "Save",      'S',11, savef, 0,
  "Load",      'L',12, loadf, 0,
  exitxt,      'X',-1, NULL,  0
};
#define N3 sizeof(items_3)/sizeof(menuitem)
```

```
/*      A teljes menurendszer leirasa:                              */
menutype menus[ ] =
{/* head.   x  y  xs  ys  itemno items    hier. last   */
  "",       9, 2, 13, N0+3, N0,  items_0,  1,   0,
  "",      35, 2, 14, N1+3, N1,  items_1,  1,   0,
  "",      61, 2, 14, N2+3, N2,  items_2,  1,   0,
  "Files",11, 6,  8, N3+3, N3,  items_3,  0,   1
};
```

Figyeljük meg, hogy a menülisták méretének meghatározását a fordító programra bíztuk: a `sizeof` operátor segítségével megkapjuk mind az egyes menülistákat tartalmazó tömbök helyfoglalását byte-okban, mind a `menuitem` típus méretét; ezek hányadosa adja meg a menülista tömbök logikai méretét (azaz azt, hogy hány elemű egy menülista). Ezeket a kifejezéseket `#define` makróként definiáljuk, és az így kapott kifejezéseket használjuk fel a `menus` tömb inicializálására. Ez egy igen flexibilis megoldás, ugyanis egy menülista bővítése során a `menus` tömb inicializálásakor a menüdoboz méretére és a menülista hosszára vonatkozóan automatikusan helyes adatot fog a fordító felhasználni. A `menus` tömb kitöltését legfeljebb csak akkor kell módosítani, ha egy új menülista-elem hossza nagyobb, mint az adott menüdobozhoz megadott `xs` érték.

Saját include file-ok

Menürendszerünk egy függvényekre mutató pointerekből álló tömb segítségével aktivizálja az egyes menüpontokhoz rendelt függvényeket. Ahhoz, hogy ezt a pointertömböt ki lehessen tölteni, szükség van a saját függvényeink prototípusaira. Fontos, hogy csak `int` típust visszaadó, egyetlen `int` típusú paramétert váró függvényeket illeszthetünk be a menürendszerbe. Ha ettől eltérő rutinjaink vannak, akkor azokat "fejeljük meg" úgy, hogy ennek a követelménynek eleget tegyenek. Ezeket a függvényeket vagy úgy deklaráljuk, ahogy azt az adatstruktúra leírásakor tettük, vagy egy include file-ba foglajuk a deklarációkat. A kettő egyszerre is alkalmazható, feltéve, ha a kétféle deklaráció összhangban áll egymással. Mi most a menükezelő rendszerben történő deklarációt alkalmazzuk, és csak a menükezelő rutinok deklarációit helyezzük el a saját file-ban.

A bevezetőben említett, nem portábilis képernyőkezelő függvényeinket egy önálló `.c` file-ban érdemes tárolni, prototípusaikat szintén a függvény rpototípusokat tartalmazó include file-unkban érdemes elhelyezni.

Ezt az include file-t, amit például `myfunc.h`-nak nevezhetünk majd a menükezelő rendszert tartalmazó `.c` file fogja behívni a

```
#include "myfunc.h"
```

preprocesszor utasítással.

Érdemes a menükezelő rendszerünk által használt különféle szimbólumokat is – például egyes speciális billentyűk kódjainak szimbólikus neveit, mint például `RIGHT` ami a → billentyű kódjának, `LEFT`, `UP`, `DOWN`, `ESC`, `BEGIN`, `END`, `HELP` rendre a ←, ↑, ↓, *Esc*, *Enter*, *Home*, *End* és az *F1* billentyű kódjának felel meg az IBM PC-n – egy szimbólum file-ba foglalni. Legyen ennek a file-nak a neve például `mysymb.h`. Ezt a file-t szintén az `#include` direktívával építhetjük be a rendszer minden egyes `.c` file-jába. (Megjegyezzük, hogy ez a file akár `#define`-nal deklarált makrószerű konstansokat tartalmazhat, akár `const`-ként definiált konstansok deklarációit tartalmazhatja – az itt közölt programrészletek szempontjából ez lényegtelen. Egy másik megjegyzés az egyes billentyűkhöz rendelt kódokra vonatkozik: A speciális billentyűkhöz célszerű 128-nál nagyobb kódokat rendelni. Így a billentyűzet kezelő függvény által visszadott billentyűkódok közül könnyen kiszűrhetők a közvetlen ASCII karakterkódok. A menükezelő rendszerben ezzel a feltételezéssel élünk.

2.13.2 A menükezelő rendszer listája

```
/*********************************************************
* File:          menu.c                                  *
* Tartalom:      Menukezelo mintaprogram                 *
*********************************************************/

#include <stdio.h>  /* Standard i/o csomag                    */
#include <string.h> /* Sztring- es memoriakezelo rutinok      */
#include <stdlib.h> /* Altalanos celu standard fuggvenyek     */
#include <ctype.h>  /* Karakterkezelo makrok                  */

#include "myfunc.h" /* Sajat fuggvenyek prototipusai          */
#include "mysymb.h" /* Szimbolumok (spec. billentyuk kodjai)*/

/* ================================================== */
/*                    Tipusdeklaraciok                */

typedef int intfunc(int);/* int-et visszaado, 1 int-et varo *
                          * fuggvenytipus                  */

typedef intfunc *fad;    /* intfunc tipusu  tarolasi  egy-  *
                          * segre mutato tipus              */

typedef struct
{
   char *text;          /* A menupont azonosito szovege      */
   char key;            /* A menupontot kivalaszto betu      */
   int  helpindex;      /* A menuponthoz rendelt help-kod    */
   fad  function;       /* A menuponthoz tartozo fv-re mutat */
   int  param;          /* A '*function' fuggveny parametere */
} menuitem;

typedef struct
{
  char     *header;  /* A menu fejlecszovegere mutat          */
  int      x;        /* A menudoboz bal felso sarkanak        */
  int      y;        /* x es y koordinatai, valamint          */
  int      xs;       /* a menudoboz x es                      */
  int      ys;       /* y iranyu merete.                      */
  int      itemno;   /* A menupontok szama                    */
  menuitem *items;   /* A menupontok listajara mutat.         */
```

```
  int      hierarch; /* Ha 1, kozvetlenul a fomenu hivja    */
  int      lastitem; /* Utoljara kivalasztott pont szama    */
} menutype;

/* ========================================================= */
/*       Tarolasi egysegek deklaracioi, definicioi:          */

static char exitxt[~] = "eXit";
                     /* Majd sokszor kell ez a sztring.     */

/* Kulso fuggvenyek deklaracioja                            */

extern intfunc data,  r_data, w_data, statf,
              regr,  linf, barf,
              save,  load;

/* A a menukezelo fuggveny prototipus erteku deklaracioja   */

intfunc menu;       /* Elore hivatkozashoz                  */
intfunc dir, shell; /* Tovabbi fv-ek elore hivatkozshoz     */

/* Az egyes menulistak (items_0 .. items_3) es a menuk:     */

menuitem items_0[ ] =
{ /* text      key hlp func. param.   */
  "Directory", 'D', 1, dir,   0,
  "Os shell",  'O', 2, shell, 0,
  "File",      'F', 3, menu,  3,/*a 3.sz. menu almenu lesz */
  exitxt,      'X',-1, NULL,  0 /*-1-es parameter: exit    */
};                                /* Tombmeret:             */
#define N0 sizeof(items_0)/sizeof(menuitem)

menuitem items_1[ ] =
{
  "Default",   'D', 4, data,  7,
  "Read data", 'R', 5, r_data,1,
  "List data", 'L', 6, w_data,2,
  "Statistics",'S', 7, statf, 3,
  exitxt,      'X',-1, NULL,  0
};
#define N1 sizeof(items_1)/sizeof(menuitem)
```

```
menuitem items_2[ ] =
{
  "Regression",'R', 8, regr,  4,
  "Plot",      'P', 9, linf,  5,
  "Bar",       'B',10, barf,  6,
  exitxt,      'X',-1, NULL,  0
};
#define N2 sizeof(items_2)/sizeof(menuitem)

menuitem items_3[ ] =
{
  "Save",      'S',11, savef, 0,
  "Load",      'L',12, loadf, 0,
  exitxt,      'X',-1, NULL,  0
};
#define N3 sizeof(items_3)/sizeof(menuitem)

/*      A teljes menurendszer leirasa:                          *
menutype menus[ ] =
{/* head.  x  y  xs ys itemno items hier. last  */
  "",       9, 2, 13, N0+3, N0, items_0, 1, 0,
  "",      35, 2, 14, N1+3, N1, items_1, 1, 0,
  "",      61, 2, 14, N2+3, N2, items_2, 1, 0,
  "Files",11, 6,  8, N3+3, N3, items_3, 0, 1
};
```

/*
Mivel a főmenünek semmi más funkciója nincs, mint a `menu` *függvénynek átadni a vezérlést a megfelelő menüindexszel, komolyabb adatstruktúrákat nem definiáltunk a számára. Csak az alábbiakra van szükség a főmenühöz:*
*/

```
static char main_header[ ] =  /* A fomenu fejlecszovege      */
         " Highly Portable Menu System ";

static char options[ ]=/*Az egyes menuk kivalaszto gombjai  */
         "FDP";       /*Sorrendjuk ugyanaz, mint az alab-  */
                      /*bi sztring-tomb elemeinek sorrendje*/
```

/*
Az `options` *sztring hossza adja meg, hogy a* `menus` *tömb hányadik eleméig tekintjük a menüket a főmenü részeinek.*
*/

```
static char *headers[ ]= { "File", /* A fomenube felvett    */
                           "Data", /* menuk fejlec szove-   */
                           "Plot"  /* gei.                  */
                         };
static int  mainselect = 0; /* Az utoljara kiv.fomenu elem  */
static char buffer[81];     /* Ide generaljuk a fomenut     */
static int  xp,yp,j;        /* Segedvaltozok a rutinokhoz   */
static char inpbuff[256];   /* Altalanos input buffer       */
```

/*
A magyarázatok és deklarációk után következzenek maguk a függvények! A menükezelő rendszert úgy hoztuk létre, hogy programunk `main`*-je csak ilyen rövid legyen:*
*/

```
/*************************************************************/
void main(void) /* Ez tetszoleges program 'main'-je lehet   */
/*************************************************************/
{
  displ_ini(); /* A kepernyokezelo rendszer inicializalasa  */
  main_frame();/* Keretrajzolas a fomenuhoz: mint az IDE    */
  main_menu(0);/* Fomenu. Addig fut, amig ESC-pel
                  ki nem szallnak belole                     */
  displ_end(); /* A kepernyo alapallapotanak
                  helyreallitasa                             */
  exit(0);     /* OK hibakod visszadasa az operacios
                  rendszernek                                */
}
```

/* *Most tekintsük magát a menükezelő rutincsomagot!* */

```
/*************************************************************/
int menu(int index)/*Az aktualisan hasznalando menu indexe  */
/*
```

Funkció:
A függvény a `menus[index]`*-ben adott menüt megjeleníti a képernyőn. Az egyes menüpontokat a menüleírás szerinti dobozban jeleníti meg. A* `menus[index].lastitem` *indexű menüpont kiemelve látszik a képen. A kiemelt menüpontot a* ↑ *és* ↓ *kurzorvezérlőkkel változtathatjuk. Ha leütjük az Enter billentyűt, akkor a kiemelt szinű menüpont függvényét hivjuk meg, ha pedig valamelyik menüponthoz rendelt nagybetűt ütjük le a billentyűzeten, akkor az illető menüpont függvénye lesz aktivizálva a* `menus[index].items[selected].param` *parameterrel, ahol* `index` *a kiválasztott menüpont indexe. Amint a meghívott függvény visszaadja a vezérlést,*

a `menu` *szubrutin regenerálja az aktuális menülistát a keretezett dobozban. Ha* `menus[index].hierarch == 1` *akkor a* `menu` *függvény visszatérési értéke*

- `RIGHT` *ha a* $\rightarrow$ *kurzorvezérlő gombot nyomták meg,*
- `LEFT` *ha a* $\leftarrow$ *kurzorvezérlő gombot nyomták meg.*

Minden egyéb esetben a visszatérési érték 0, tehát amikor

- *az ESC gombot nyomták meg (kilépés a* `menu` *függvényből),*
- *olyan menüpontot választottak ki, amelynek a* `helpindex`*-e* `-1`

```
*************************************************************/
{
    int i,              /* A menupontok szamat tesszuk bele    */
        l,              /* for-ciklushoz ciklusvaltozo         */
        exit,           /* Kilepest jelzo flag                 */
        par,            /* A kivalasztott fv. parametere       */

        cmd;            /* A vezerlo-karakternek               */

    /* ......... E L O K E S Z I T E S E K ............... */

    j = menus[index].lastitem;/* j-ben az aktualis index       */
    i = menus[index].itemno;
    if (!i) return 0; /* Nulla meretu menuvel nem torodunk     */

    menu_regen(index,1);/* A menut kiiratjuk a kepernyore      */
    exit = FALSE;       /* A kilepest jelzo flag kezdoerteke   */

    /* .............. F O   C I K L U S ................... */

    while (! exit)      /*Addig tart,amig exit igaz nem lesz    */
    {
      cmd = 0;          /* Kezdetben ures parancs              */
      while (!(cmd == SELECT || cmd == CR))
      {
         cmd = getkey(); /* VT100-on ketfele ENTER van,        */
         switch(cmd)     /* ezert van CR is es SELECT is.      */
         {
           case BEGIN:
              o_gotoxy(xp,yp+j); /* HOME-ot nyomott           */
```

```
            printf("%s",menus[index].items[j].text);
            j = 0;
            o_gotoxy(xp,yp);
            highlight(EMPHAS,menus[index].items[j].text);
            break;
         case END:
            o_gotoxy(xp,yp+j);  /* END-et nyomott          */
            printf("%s",menus[index].items[j].text);
            j = i-1;
            o_gotoxy(xp,yp+j);
            highlight(EMPHAS,menus[index].items[j].text);
            break;
         case UP:                /* 'fel' nyil             */
            {
              o_gotoxy(xp,yp+j);
              printf("%s",menus[index].items[j].text);
              if (j > 0) j--; else j = i - 1;
              o_gotoxy(xp,yp+j);
              highlight(EMPHAS,menus[index].items[j].text);
            }
            break;
         case DOWN:              /* 'le' nyil              */
            {
              o_gotoxy(xp,yp+j);
              printf("%s",menus[index].items[j].text);
              if (j < i-1) j++; else j = 0;
              o_gotoxy(xp,yp+j);
              highlight(EMPHAS,menus[index].items[j].text);
            }
            break;
         case HELP:              /* F1-et nyomtak          */
   menus[index].lastitem = j;
            menu_help(menus[index].items[j].helpindex);
            if (menus[index].items[j].helpindex >= 0 &&
                menus[index].y + menus[index].ys > 11)
            menu_regen(index,0);
            break;
         case ESC:               /* ESC-et nyomtak         */
            exit = 1;
            cmd = SELECT;
            break;
         case LEFT:
```

```
        case RIGHT:
           /* Ha 'main_menu' hivta 'menu'-t, akkor a
              'jobbra', 'balra' nyilak eseten a menut to-
              roljuk, es a nyil-gomb kodjat visszaadjuk.
              Igy a fomenu a roll-in menut felvaltja egy
              masikkal:                                  */
           if (menus[index].hierarch == 1)
           {
             menu_remove(index);
             return cmd;
           }
           default:
      /* Kilepunk, ha dedikalt gombot nyomtak        */
           if (cmd < 128)
           {
             cmd = toupper(cmd);
             for(l = 0; l < i; l++)
             {
               if (menus[index].items[l].key == cmd)
               {
                 o_gotoxy(xp,yp+j);
                 printf("%s",menus[index].items[j].text);
                 cmd = SELECT;
                 j = l;
                 break;
               }
             }
           }
           break;
       } /* ............... end switch ................... */
     } /* ................. end while ..................... */
     if (! exit)
     {
       exit = (menus[index].items[j].helpindex == -1);
     }

/*
Ezen a ponton már eldőlt, hogy ki akarunk-e lépni. Ha nem, akkor viszont
tudjuk, hogy melyik menüpont függvényét kell aktivizálni:
*/

     if (! exit)
     { /* Az 'eXit' pontnak mindig -1 helpindexe legyen!   */
```

```
        /* .... A kivalasztott fuggveny aktivizalasa:....   */
        /*      (j indexeli a kivalasztott fuggvenyt)       */

        o_gotoxy(xp,yp+j);
        highlight(EMPHAS,menus[index].items[j].text);
        menus[index].lastitem = j;
        par = menus[index].items[j].param

        (*menus[index].items[j].function)(par);

        menu_regen(index,0);  /* A menu-box regeneralasa    */
      }
      else
      {
         menu_remove(index);
      }
    }
    return 0;
}
/*************************************************************/
void menu_regen(int index, /* A regeneralando menu indexe   */
                int rem)   /* TRUE: torolni kell a dobozt   */

/*
Funkció:
A menus[index] menü regenerálása (újra rajzolja a dobozt, kiírja a menülistát és kiemelő szinnel nyomtatja az utoljára kiválasztott menüpontot.)
Ha rem == 1 akkor a menü által elfoglalt képernyőterületet törli a menülista kiírása előtt, ha rem == 0, akkor nem töröl.

*************************************************************/
{
    int i,k,l,m,n,xx,yy;
    int x1,x2;
    xp = menus[index].x;     /* Pozicio, meret elovetele     */
    yp = menus[index].y;
    i  = menus[index].itemno;
    xx = menus[index].xs;
    yy = menus[index].ys;
                             /* Dobozrajzolas                */
    box_draw(menus[index].header,xp,yp,xx,yy,rem);
```

```
    xp += 2;
    yp += 2;
    for (k = 0; k < i; k++) /* A menulista megjelenitese    */
    {
       o_gotoxy(xp,yp+k);
       if (k == menus[index].lastitem)
       {
           highlight(EMPHAS,menus[index].items[k].text);
           j = k;
       }
       else
           printf("%s",menus[index].items[k].text);
    }
}
/*******************************************************/
void menu_remove(int index) /* A torlendo menu indexe       */

/*
Funkció:
A menus[index] menü törlése a képernyőről

*******************************************************/
{
    int xx,yy,x1,y1;
    x1 = menus[index].x;
    y1 = menus[index].y;
    xx = menus[index].xs;
    yy = menus[index].ys;
    box_delete(x1,y1,xx,yy);
}
/*******************************************************/
void box_draw(char* header,    /* ->a doboz fejlec-szevege  */
              int   xp, int yp,/* a doboz pozicioja,        */
      int   xs, int ys,/* merete                    */
      int rem)  /* 1, ha torles kell, egyebkent 0   */

/*
Funkció:
Egy xs, ys méretű dobozt rajzol az xp, yp pozicióba. A keret felső részének
közepére a header fejlécet írja ki. Ha rem == 1, akkor a doboz rajzolása
előtt törli a doboz által elfoglalandó területet.

*******************************************************/
```

```
{
    int             l,n,xx,yy;
    int             x1,x2;
    l  = strlen(header);  /* A fejlec hossza                  */
    xx = xs-2;            /* Egyeb adatok elokeszitese        */
    x1 = (xx - l)/2;
    x2 = xx - (x1 + l);
    yy = ys-2;
    if (rem) box_delete(xp,yp,xs,ys);

    o_gotoxy(xp,yp);      /* A legfelso sor a fejleccel       */
    printf("%c",UPLEFT);
    for (n = 0; n < x1; n++) printf("%c",HORIZ);

    highlight(REVERSE BRIGHT,header);

    for (n = 0; n < x2; n++) printf("%c",HORIZ);
    printf("%c",UPRIGHT);
    yp++;
    for (n = 0; n < yy; n++) /* Maga a doboz                  */
    {
      o_gotoxy(xp,yp+n);
      printf("%c",VERT);
      o_gotoxy(xp+1+xx,yp+n);
      printf("%c",VERT);
    }
    o_gotoxy(xp,yp+yy);       /* A doboz legalso sora         */
    printf("%c",DOWNLEFT);
    for (n = 0; n < xx; n++) printf("%c",HORIZ);
    printf("%c",DOWNRIGHT);

}
/*****************************************************/
void box_delete(int xp, int yp, /* Egy dobozt torol      */
                int xs, int ys) /* Pozicio, meret        */

/*
Funkció :
Egy xs, ys méretű dobozt töröl az xp, yp pozicióról.
*****************************************************/
{
    int  n, m;
```

```
    for (n = ys-1; n >= 0; n--)
    {
        o_gotoxy(xp,yp+n);
for (m = 0; m < xs; m++) putc(' ');
    }
}

/*******************************************************/
void menu_help(int index) /* A menupont help-indexe          */

/*
```

Funkció :
Az `index` *által meghatározott help-szöveget kikeresi egy help-file-ból, és kiírja a képernyőre. A kiíráshoz egy 7 soros ablakot nyit, a szöveget 7 soronként írja ki. Ha van még kiirandó szöveg, akkor a* More ... *üzenet után egy billentyűleütésre vár, ha nincs, akkor a* Press any key ... *üzenet után törli a képernyőről a help-dobozt, és visszatér. A help-file formátuma a következő:*

```
index_i  n_i
sor_1_i
sor_2_i
 ...
sor_n_i
index_j  n_j
 ...
```

ahol `index_i` *az i-edik help-index,* `n_i` *az ehhez az indexhez tertozó help-szöveg sorainak a száma, valamint* `sor_1_i`, ... `sor_n_i` *a help-szöveg egyes sorai. Formátum hiba, vagy file vége esetén szintén hibajelzés történik.*

```
*******************************************************/
{

static char hunex[] = "Unexpected end of the help-file!!!";
#define YP 24

    FILE *fp;
    int i,j,k,err;

    if (index < 0) return; /* Negativra visszater          */
```

```
box_draw(" HELP ",2,11,76,9,1);/* Help-box rajzolasa    */
fp = fopen(helpdat,"r");       /* Help-file megnyitasa  */

if (fp == NULL)  /* Ha nem letezik a file, hibajelzes   */
{
  o_gotoxy((80-(21+strlen(helpdat)))/2-1,16);
  printf("Help-file %s not found!",helpdat);
  goto helpend1;
}
i = -1;
while (i != index)  /* Help-index keresese a file-ban   */
{
   err = fscanf(fp,"%d%d",&i,&j);
   if (err == EOF)  /* Nem talaljuk: hibajelzes         */
   {
      o_gotoxy(19,16);
      printf("No help is available for this menu item!");
      goto helpend0;
   }
   if (err != 2)
   {
      o_gotoxy((79-(31+strlen(helpdat)))/2,16);
      printf("Format error in the %s help-file!",helpdat);
      goto helpend0;
   }
   if (i != index)/* Ha meg nem talalja, tovabb olvas.  */
   {
     for (; j >= 0; j--)
     {
       if (NULL == fgets(inpbuff,74,fp))
       {
          o_gotoxy((79-strlen(hunex))/2,16);
          printf(hunex);
          goto helpend0; /* A 'goto'-t legfeljebb igy   */
       }                 /* hasznaljuk!                 */
     }
   }
}
for (k = i = 0; i < j; i++)
{
   if (NULL == fgets(inpbuff,74,fp))
   {
```

```
            o_gotoxy((79-strlen(hunex))/2,16);
            printf(hunex);
            goto helpend0;
         }
         o_gotoxy(4,12+k);
         printf(inpbuff);
         k++;
         if (k == 7)/*Megvan a helpszoveg. 7-esevel kiirjuk:  */
         {
            o_gotoxy(66,YP);
            highlight(BRIGHT,"More ...");
            bell();
            err = getkey();
            o_gotoxy(66,YP);
            printf("         ");
            if (err == ESC)/* ESC-re kiszallunk                */
            {
               fclose(fp);
               box_delete(2,11,76,9);
               return;
            }
            box_draw(" HELP ",2,11,76,9,1);
            k = 0;
         }
      }
      helpend0:   /* Minden befejezeskor ezeket a muveleteket */
      fclose(fp); /* kell elvegezni, tehat  takarekos  meg-   */
      helpend1:   /* oldas a 'goto' hasznalat. Csinjan ban-   */
      press_key();/* junk az ilyennel, hogy olvashato marad-  */
      box_delete(2,11,76,9); /* jon a programunk!             */
}

/*************************************************************/
void main_frame(void)

/*
Funkció :
Keretet rajzol a főmenünek. Ha valamelyik függvény törli az egész képernyőt, akkor main_frame meghívásával helyreállíthatja azt.

*************************************************************/
{
  erase();
```

```
  box_draw(main_header,0,0,80,23,0);/* Main box fejleccel   */
  main_menu(1);                     /* Fomenu statusz-sora  */
}
/*********************************************************/
void main_menu(int stl)/*Ha 1, akkor a statusz-sort kiirja  */

/*
```

Funkció :
A menükezelő rendszer fő rutinja, ezt kell a `main`*-ből meghívni. A* `menus` *tömbből annyi menüt kezel közvetlenül, amennyi az* `options` *sztring hossza. A menü-opciókat a képernyő második, kivilágított sorában jeleníti meg. Egy menüpont a szokásos módon választható (kurzorral kiemelés, majd Enter, vagy a kezdő betű leütése). Ha egy almenü él (azaz látszik a képernyőn), akkor a* ←, *illetve* → *nyilakkal a szomszédos menüre válthatunk.*

```
*********************************************************/
{

  int i,j,k,l,
      posinc,
      hno,xp,
      cmd,flag;

  /* 'buffer'-ben lesz az inverzen kiirando statusz-sor     */

  hno = sizeof(headers)/sizeof(char*);
  posinc = 78/hno;

  xp = posinc/2;

  if (stl)
  {
    for (i = 0; i < 78; buffer[i++] = ' ')
    ;
    for (j = 0; j < hno; j++)
    {
        l = strlen(headers[j]);
        for(k = 0; k < l; k++)
           buffer[xp+j*posinc+k] = *(headers[j]+k);
    }
    buffer[78] = '\0';
    o_gotoxy(1,1);
```

```
    highlight(REVERSE,buffer);
  }

  /* A kivalasztott menut normal modon jelenitjuk meg:       */

  i = mainselect;
  xp++;
  if (stl)
  {
    o_gotoxy(xp+i*posinc,1);
    printf(headers[i]);
    return;
  }

  /* A fo parancs-ciklus. Csak ESC-re lephetunk ki belole.  */

dontquit:           /* Ide ugrunk, ha megse lepunk ki.      */
  flag = cmd = 0;
  while (cmd != ESC)
  {   if (! flag) cmd = getkey();
      flag = 0;
      switch (cmd) /* Nincs elo almenu. Kurzorvezerlok      */
      {            /* feldogozasa, status-sor modositasa    */
         case RIGHT:
                o_gotoxy(xp+i*posinc,1);
                highlight(REVERSE,headers[i]);
                if (i < hno-1) i++; else i = 0;
                o_gotoxy(xp+i*posinc,1);
                printf(headers[i]);
                break;
         case LEFT:
                o_gotoxy(xp+i*posinc,1);
                highlight(REVERSE,headers[i]);
                if (i) i--; else i = hno-1;
                o_gotoxy(xp+i*posinc,1);
                printf(headers[i]);
                break;
         case SELECT:
         crselect:
           /* Kivalasztottak egy almenut. Megje-    */
   /* gyezzuk indexet 'mainselect'-ben      */
```

```
                mainselect = i;
                flag = menu(i);/* A 'menu' rutin behivasa   */
                switch (flag)  /* Mi a visszateresi ertek? */
                {
                   case RIGHT: /* Stat.sor modositas ...    */
                          o_gotoxy(xp+i*posinc,1);
                          highlight(REVERSE,headers[i]);
                          if (i < hno-1) i++; else i = 0;
                          o_gotoxy(xp+i*posinc,1);
                          printf(headers[i]);
                          break;
                   case LEFT:
                          o_gotoxy(xp+i*posinc,1);
                          highlight(REVERSE,headers[i]);
                          if (i) i--; else i = hno-1;
                          o_gotoxy(xp+i*posinc,1);
                          printf(headers[i]);
                          break;
                }
                l = strlen(headers[i]);
                o_gotoxy(xp+i*posinc+l,1);
                break;
        default:
                if (cmd < 128) /*Kezdobetuvel valasztottak */
                {
                   cmd = toupper(cmd);
                   for (l = 0; l < hno; l++)
                   if (cmd == options[l])
                   {
                     o_gotoxy(xp+i*posinc,1);
                     highlight(REVERSE,headers[i]);
                     i = mainselect = l;
                     o_gotoxy(xp+i*posinc,1);
                     printf(headers[i]);

    /* Ugy teszunk, mintha 'nyil+Enter'-rel
       valasztottak volna.                    */
                     cmd = SELECT;
                     goto crselect;
                   }
                }
                break;
```

```
      }
  }

 /* Az ESC-pel valo kilepes szandekat megerosittetjuk:       */

  box_draw("",28,5,24,3,0);
  o_gotoxy(30,6);
  highlight(BRIGHT,"Are you sure? (y/n) ");
  cmd = yesno();
  box_delete(28,5,24,3);
  o_gotoxy(1,1);
  if (!cmd) goto dontquit; /*Nem lep ki, vissza az elejere  */
  erase();
}
*************************************************************/

/*
Két gyakori funkció portábilis megvalósítását találjuk itt. Ezek az aktív
könyvtár tartalmának kiiratása a képernyőre, illetve az operációs rendszer
parancs-értelmező burkának (command shell) az aktivizálása. Mindkettőt
a system függvény segítségével oldjuk meg. A system argumentuma egy
operációs rendszernek szóló parancsot tartalmazó sztring. Ezek a mi ese-
tünkben egy-egy #define makróként lettek megadva, így azok operációs
rendszertől függő feltételes fordítással megfelelően beállíthatók. Tahát a
system függvénynek (process.h) átadandó operációs endszer parancsok:
*/


#ifdef __MSDOS__
#define  DIRSTR  "dir /w/p"
                      /* A burok (shell) 'dir' parancsa      */
#define  SHELL   "COMMAND.COM"
                      /* Maga az operacios r. burok (shell)  */
#endif

/*
A fenti sztringeket csak a DOS-ban adhatjuk át a system-nek, ezért hasz-
náltuk az #ifdef __MSDOS__ fordításvezérlő direktívát. UNIX-ban a meg-
felelő sztringek értéke rendre "ls -C|more", illetve "sh" lenne.
*/

/*************************************************************/
int dir(int d)  /* Az aktiv konyvtar tartalmat nezzuk meg   */
```

```
                /* d: Dummy parameter                              */
/*****************************************************************/
{
   displ_end();
   system(DIRSTR); /* Az op. rendszer DIR parancsat kerjuk.
                      Ennel lehetne hatekonyabb  megoldast
                      is talalni, de ez igy portabilis.     */
   displ_ini();
   press_key();
   erase();
   main_frame();
   return d;
}
/*****************************************************************/
int shell(int d)/* A parancsertelmezo burok hivasa               */
                /* d: Dummy parameter                              */
/*****************************************************************/
{
   displ_end();
   printf("\nType exit to return!\n\n");
   system(SHELL);
   displ_ini();
   erase();
   main_frame();
   return(x);
}
```

A saját include file tartalma

A következő lista a **myfunc.h** include file javasolt tartalmát mutatja be:

```
/*****************************************************
* File:          myfunc.h                            *
* Tartalom:      Menukezelo mintaprogram fuggveny proto-  *
*                tipusai: elorehivatkozasokhoz, illetve a  *
*                kepernyokezelo rendszer hasznalatahoz.    *
*****************************************************/

/* ================================================== */
/*       A menukezelo rendszer fuggvenyeinek deklaracioi    */

void main_menu(int stl);     /* Fomenu-rutin. Ezt kell a main-
                                bol meghivni                */
void main_frame(void);       /* A fomenuhoz keretet rajzol a
                                kepernyore                  */
int  menu(int index);        /* Ez a menurutin: az adott sor-
                                szamu menut kezeli          */
void menu_regen(int index); /* Az adott sorszamu menut rege-
                                neralja a kepen             */
void menu_remove(int index);/* Az adott sorszamu menut letor-
                                li a keprol                 */
void menu_help(int index);  /* Adott menuponthoz helpet ir ki
                                egy file-bol                */
void box_draw(char *header, /* Adott fejleccel,            */
              int xp,int yp,/* adott xp,yp poxicioban,     */
              int xs,int ys,/* adott xs,ys meretben dobozt  */
              int rem);     /* rajzol, ha kell, torol alatta*/
void box_delete(int xp,     /* Adott helyrol adott meretu  */
                int yp,     /* dobozt torol                */
                int xs,
int ys);
/* ================================================== */
/*  A keprnyokezelo rutinok prototipusai magyarazatokkal    */

void o_gotoxy(int x, int y); /* Sajat pozicionalo.
                                x=0..24, y=0..79            */
void home(void);             /* A kurzort a 0,0 pozicioba
                                helyezi                     */
void erase(void);            /* Torli a kepernyot es a
                                0,0-ba pozicional           */
```

```
void displ_ini(void);          /* Bekapcsolja a kepernyo-
                                  kezelo rendszert, torol    */
void displ_end(void);           /* Kikapcsolja a kepernyo-
                                  kezelo rendszert, torol    */
void cursor_left(int n);       /* A kurzort egy pozicioval
                                  balra viszi                */
void cursor_right(int n);      /* A kurzort egy pozicioval
                                  jobbra viszi               */
void cursor_up(int n);         /* A kurzort egy pozicioval
                                  feljebb helyezi            */
void cursor_down(int n);       /* A kurzort egy pozicioval
                                  lejebb helyezi             */
void highlight(unsigned mode,/* 'mode' szerinti attributum-
               char* string);/* mal 'string'-et nyomtatja   */
int  read_in(char* string);    /* Egy sztringet olvas be     */

int  yesno(void);              /* y/Y/I/i/N/n (igen/nem)
                                  valaszt var. 0, ha 'nem'   */
int  input(char* string,
           int pos,int  len);/* Sor-editor. Adott pozici-
                                  on, adott hosszt edital    */
void press_key(void);          /* A 'Press a key' kiirasa u-
                                  tan gombnyomasra var       */
void bell(void);               /* Egy BEL karaktert kuld az
                                  stdout-ra: igy beep-el     */
int  getkey(void);             /* A billentyuzetet kezeli,
                                  ASCII-t, illetve #define
                                  erteket ad vissza (pl. UP) */
```

3 Fejezet

Programozás C++-ban

A C++ nyelv a C programozási nyelv egy továbbfejlesztett változata, így néhány részlettől eltekintve, felülről kompatibilis a C-vel. A C nyelv lehetőségein túlmenően, a C++ igen flexibilis és hatékony eszközöket nyújt a programozónak új adatstruktúrák definiálására. A C++ lehetővé teszi, hogy a programozó munkáját olyan jól kezelhető részekre oszthassa fel, amelyek szorosan kötődnek az alkalmazói program alapkoncepciójához. Másképp fogalmazva: a C++ megteremti annak a lehetőségét, hogy egy problémát adattípusokra, és az adott adattípusokhoz szorosan hozzárendelt műveletekre képezhessünk le. Így a valósághoz közel álló objektumokat definiálhatunk a C++ programban. Ezek az objektumok aztán kényelmesen és biztonságosan használhatók olyan kontextusban is, amikor típusuk a program fordításakor még nem állapítható meg. Ezt a fajta programozási technikát *objektum-orientált programozásnak* (röviden OOP-nek) nevezzük. Ha jól használják, az OOP rövidebb, áttekinthetőbb, és könnyebben karbantartható programokat eredményez, mint a "hagyományos" programozási stílus.

A C++ alapelemei az *osztályok* (*classes*). Egy *osztály* nem más, mint egy felhasználó által definiált új típus, amely a szükséges adatstruktúrát, és az adott struktúrájú adatokkal végezhető műveleteket definiálja. Az osztályok használata lehetővé teszi az információrejtést (tehát azt, hogy bizonyos dolgokról csak a használatukhoz szükséges ismereteket tesszük mások által is hozzáférhetővé), garantálja az adatok inicializálását, implicit típuskonverziót biztosít a felhasználó által definiált adattípusok esetében, az egyes operátorokhoz újabb jelentést rendelhetünk általuk, stb. C++-ban sokkal hatékonyabb eszközök állnak rendelkezésre a modularitás kidomborítására és a típusellenőrzésre, mint a C-ben. További olyan bővítéseket is tartalmaz

a C++, amelyek nincsenek kapcsolatban az objektum-orientált programozással. Ilyenek például az ún. *inline* függvények, a cím szerint átadható függvényparaméterek, az inicializált függvényparaméterek, stb.

A C++ megtartja a C-nek azt a tulajdonságát, hogy igen hatékonyan kezeli a hardver közeli adattípusokat (bitek, byte-ok, szavak, címek), így ez az új nyelv továbbra is jól felhasználható rendszerprogramozói feladatok megoldására.

A C++ nyelv tervezői elsődlegesnek tekintették a C egyszerűségének megőrzését, és a C-vel való kompatibilitást, valamint a régi C szintaxisának tisztábbá tételét. A C++-ban nincsenek magas szintű adattípusok és hozzájuk rendelt műveletek. Például nem létezik a C++-ban mátrix típus és mátrix invertáló operátor. Ha a felhasználónak szüksége van ilyen típusra, a nyelv lehetővé teszi annak definiálását. Valójában a C++ programozói tevékenység alapvető, lényegi részét a megfelelő adattípusok definiálása teszi ki. Egy jól megtervezett felhasználói adattípus nem abban különbözik egy beépített típustól, hogy hogyan lehet használni, hanem csak abban, hogy miképp van definiálva. Az objektumok típusának ismeretében a fordítóprogram megfelelően tudja kezelni a belőlük alkotott kifejezéseket, míg a hagyományos C-ben a programozónak kín-keservvel kell leírnia minden egyes művelet végrehajtási módját. A típusok pontos ismerete azt is elősegíti, hogy már programfordításkor kiderüljenek olyan hibák, amelyek egy hagyományos C program esetében csak a tesztelés során találhatók meg.

Ebben a részben – ahogy a 2. fejezetben áttekintettük a BORLAND C++ implementáció ANSI C kompatibilis elemeit – ismertetjük a nyelv AT&T C++ 2.0 kompatibilis részeit. A C++-ra vonatkozó alapreferenciaként ajánljuk Bjarne Stroustrup könyvét [3]. Fel kell azonban hívnunk az olvasó figyelmét arra, hogy ez a könyv "csak" az 1.0-ás verziójú C++ egzakt referenciája. Mivel az AT&T a 2.0-ás C++ változatban a korábbi verziót tovább bővítette, mi olyan nyelvi elemekről is említést teszünk, amelyekről Stroutstrup fenti könyve (értelemszerűen) nem szól. A C++-ra vonatkozó legfrissebb referenciaként Stroustrup egy újabb munkáját ajánljuk [4]. A C++ nyelv BORLAND C++ implementációjának teljes referenciáját az eredeti programdokumentáció *Programmer's Guide* című kötete [10] tartalmazza.

3.1 Új elemek a C++-ban

Mielőtt komolyan foglakoznánk az objektum-orientált programozás alapjaival és azoknak C++-beli megvalósításával, tekintsük át a C++ által nyújtott azon újdonságokat, amelyek bővítést jelentenek a hagyományos ANSI C-hez képest, de még nem igénylik az OOP gondolkodásmódot.

3.1.1 Alternatívák a `#define` direktíva helyett

A 2.3-as részben megismerkedtünk a `#define` preprocesszor utasítással. Ez a direktíva három célt szolgál:

- feltételes fordítást vezérlő szimbólumokat definiálunk vele az előfeldolgozó `#if ... #elif ... #else ... #endif`, illetve `#ifdef`, `#ifndef` szerkezetei számára,
- programjaink szám-, vagy szövegkonstansaihoz, illetve konstans-kifejezéseihez szimbólikus neveket rendelhetünk, ezáltal növelve a programkód rugalmasságát és olvashatóságát, és végül
- függvény-jellegű makrókat definiálhatunk segítségével.

Az utóbbi két alkalmazást célszerű elválasztani a fordításvezérlő funkcióktól. Ahhoz, hogy ezt megtehessük, a C++ két lehetőséget kínál.

Az egyik a `const` kulcsszó használata. (Megjegyzendő, hogy a `const` kulcsszó az ANSI C-nek is része, lásd 2.4.6-os szakaszt.)

Minden olyan konstanst, amit nem fordításvezérlésre akarunk használni, célszerű a `const` deklarációjú valtozóknak kezdeti értékül adni. Ezek után a `const` deklarációjú váltzókat ugyanúgy használhatjuk, mint a jól megszokott `#define` konstansokat, azzal a nem mellékes különbséggel, hogy mentesülünk a `#define` konstansok feldolgozásokor történő egyszerű szöveghelyettesítés esetleges kellemetlen mellékhatásaitól. Ezáltal biztonságosabbá válhatnak forrásprogramjaink.

A `const` módosító szóval deklarált változók mellett a felsorolt típusú (`enum`) változók alkalmazása is hatékony alternatíva a `#define` direktíva helyett. C++ az `enum` típust kicsit másképp értelmezi, mint a hagyományos C. A különbség az, hogy míg a hagyományos C az `enum` és `int` típusokat kompatibilisnek tekinti, addig a C++ a különböző `enum` típusokra vonatkozólag is szigorú típusellenőrzést végez. Így például egy `enum` típusú változónak nem adhatunk `int` típusű értéket.

A `#define`-nal történő makródefiníciókkal szemben hatékony alternatívát jelentenek az ún. *inline* (sorok közötti) függvények. Röviden: egy inline függvény elég kis terjedelmű ahhoz, hogy helyben (*in situ*), 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 implicit inline deklarációról az ún. függvénymezők kapcsán egy későbbi fejezetben lesz szó.)

Tehát 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. Végül az explicit inline függvény-definícióra álljon itt egy példa:

```
inline int abs(int x)
{
   return (x < 0) ? -x : x;
}
```

3.1.2 Cím szerint nyilvántartott típusú, vagy referencia típusú változók

A 2.9.3-as részben, a hagyományos C nyelv ismertetésénél említettük, hogy a függvények érték szerint veszik át az aktuális paramétereiket. (A tömböket ebből a szempontból tekintsük olyan pointereknek, amelyek adott számú értékes adat számára lefoglalt tárterületre mutatnak – lásd a 2.9.6-os szakaszt. Igy a tömbök is belefférnek az "érték szerint" fogalmába, hiszen a mutatókat tényleg érték szerint adjuk át.) Ha egy változót cím szerint akartunk paraméterként átadni, akkor a formális paraméterlistában az adott típusra mutató pointert kellett deklarálnunk, a függvénytörzsben az indirekció operátorát kellett alkalmaznunk, és a függvény meghívásakor az aktuális paraméterlistában magunknak kellett explicit módon gondoskodnunk arról, hogy a megfelelő paraméterhelyre a megfelelő változó címe kerüljön. Ennek a dolognak az a nagy hátránya, hogy a függvénytörzsben nem különülnek el szintaktikailag az igazi tömbök és a cím szerint átadott skalárjellegű változók.

A C++-ban ilyen, és hasonló jellegű problémák áthidalására bevezették az ún. cím szerint nyilvántartott, vagy referencia típus (*reference type*) fogalmát és ehhez definiálták a `&` típusmódosító operátort. Igy például a

```
int& r;
```

deklaráció azt jelenti, hogy `r` olyan változó, amely egy egész típusú változóra vonatkozó referenciát tartalmazhat. Úgy is felfoghatjuk a dolgot, hogy egy ilyen fajta változó egy olyan konstans pointer-kifejezés, amelyre vonatkozólag automatikusan végrehajtódik egy indirekció-művelet, amikor az adott referencia típusú változóra hivatkozunk. Ez három dolgot von maga után. Az egyik, hogy a fenti példa szerinti `r` változó minden olyan helyen állhat, ahol egy `int` típusú változó is állhat, azaz egész típusú kifejezésekben

akár balérték, akár jobbérték lehet. A második, hogy a referencia típusú változókkal semmilyen művelet nem végezhető, hiszen minden hivatkozás alkalmával minden egyebet megelőz az implicit indirekció művelet (*dereference operation*, lásd a 2.5.2 alatt az egyoperandusú * operátorról leírtakat). Ezzel áll szoros összefüggésben a harmadik fontos dolog, hogy nincs értelme egy cím szerint nyilvántartott típusú változót inicializálás nélkül definiálni. Tehát csak az alábbihoz hasonló definíciónak van értelme:

```
int  ii = 0;
int& rr = ii;
```

Ekkor az `rr++;` utasítás szintaktikailag ugyan helyes, de nem az `rr` változó inkrementálódik, hanem az az `int` típusú tárterületfoglaló tárolási egység, amelyiknek a címét `rr` tartalmazza. Ez a fenti példában éppenséggel az `ii` változó. Ez az értelmezés triviális akkor, amikor az inicializáló kifejezés egy balérték, de nem kötelező, hogy az inicializátor balérték legyen, sőt, az sem kötelező, hogy a referncia típus alaptípusába tartozzon. Ilyen esetekben

a) először típuskonverzió hajtódik végre, ha az szükséges,

b) aztán a típuskonverzió eredménye egy ideiglenes változóba kerül,

c) és végül ennek az ideiglenes változónak a címe kerül felhasználásra az adott referencia típusú változó inicializálásához.

Tekintsük az alábbi deklarációt:

```
double& dr = 1;
```

Ez a fentiek alapján a következőképpen értelmezhető:

```
double* drp;
double temp;
temp = (double)1;
drp = &temp;
```

A használat során a `(*drp)` kifejezés egyenértékű a `dr`-rel.

A referencia típus igazán kellemes felhasználási területe a bevezetőben is említett függvényparaméter-deklaráció. Ez azt jelenti, hogy a formális paraméterlistában már lehetőségünk van arra, hogy egy paramétert ne a reá mutató pointer segítségével adjunk át cím szerint, hanem jelezzük, hogy egy olyan változóról van szó, amit cím szerint kell átadnuk (mert például kimenő paraméterként is szükségünk lesz rá), ugyanakkor a függvénytörzsben a többi, érték szerint átadott változóhoz hasonló módon – ugyanolyan szintaktikával – szeretnénk kezelni. A 2.9.3-as részben közölt egyszerű példa a referncia típus felhasználásával így néz ki:

```
void f1(long& a)
{
   a += 2L;
}
```

Ekkor az

```
alfa = 0L; f1(alfa);
```

kódrészlet szintaktikailag helyes, és hatására `alfa` értéke `2L` lesz. A különböző paraméterátadási lehetőségeket szemlélteti a következő kis program:

```
#include <stdio.h>
// *******************************************************
int any_function(int  par_by_value,  //Ertek szerinti int
                 int* use_for_arrays,//Ertek szerinti pointer
                 int& par_by_address)//Cim szerinti int
// *******************************************************
{  int   work;
   work = par_by_value;
   par_by_value *= 3;
   *use_for_arrays = par_by_address * work;
   par_by_address = work * work;
   return par_by_value;
}
// *******************************************************
main()
// *******************************************************
{
   int x = 2, y[ ] = { 1, 2 }, z = 10, w = 0;
   w = any_function(z,y,x);
   printf("%d %d %d %d\n",x,y[0],z,w);
}
```

Az `any_function` függvény harmadik paraméterét deklaráltuk referencia típusúnak, erre a típusnév (`int`) után álló `&` típusmódosító operátor utal. Tekintsük át a fenti program működését.

A `main`-ben `any_function` meghívása előtt az `x`, `y[0]`, `z` és `w` változók értéke rendre 2, 1, 10 és 0, a `w`-nek függvényhívással történő értékadás eredményeképpen (mellékhatásként) pedig `x`, `y[0]`, `z`, valamint `w` a 100, 20, 10 és 30 értékeket veszik fel; a szabványos kimeneten ezek a számok fognak sorra megjelenni. Látható, hogy az érték szerint átadott paraméter (`z`) nem változott meg, `y[0]` új értéket kapott, hiszen a reá mutató pointeren keresztül indirekt módon címezve beleírtunk, és a cím szerint átadott skalár,

x is új értékkel bír a függvényhívás után. Látható az is, hogy par_by_value és par_by_address használata szintaktikailag azonos.

Az any_function deklarációjakor a par_by_address paraméternél használt megoldás hasonlít ahhoz, amit a Pascal nyelv alkalmaz. Pascal-ban is alapértelmezés szerint érték szerint adódnak át a paraméterek; ott a VAR kulcsszóval jelezhetjük a cím szerinti paraméterátadást, ugyanakkor a FUNCTION, vagy PROCEDURE törzsében az érték, vagy cím szerint átadott paraméterek használatában nincs szintaktikai különbség.

A referencia típust függvények visszatérési típusaként is nagyon jól fel lehet használni. Gondoljunk csak a 2.12-es pontban a balértékek kapcsán deklarált char *get_buff_pos(int i) függvényre. Ennek a függvénynek balértékként való alkalmazása így nézett ki:

```
*get_buff_pos(1) = 'b';
```

Ha függvényünket char & típusú visszatérési értékkel deklaráljuk, akkor a referencia típus tulajdonságai következtében a get_buff_pos(1) kifejezés minden további nélkül érvényes balérték kifejezés lesz, tehát a fenti értékadó utasítást így írhatjuk:

```
get_buff_pos(1) = 'b';
```

Vegyük észre, hogy a referencia típusú visszatérési érték miatt a char& get_buff_pos(int i) függvény egy értékadó utasítás mindkét oldalán állhat. A referencia típusról itt elmondottak elsősorban a fogalom megértetését szolgálják. A referencia típus leges-legfontosabb szerepet azonban a felhasználó által definiált típusokra vonatkozó operátor-függvényeknek definiálásánál játssza – ezek visszatérési értéke ugyanis az adott típusra vonatkozó referencia-típusú.

Vegyük észre, hogy a referencia típust képző & operátor természetes kiegészítése a hagyományos C típusmódosító operátorainak: míg a * indirekció operátornak, a [ ] indexelő operátornak és a ( ) függvényaktivizáló operátornak volt párja a típusmódosító operátorok között, addig az 'address of' operátornak (egyoperandusú &) nem volt.

3.1.3 Inicializált függvényparaméterek

Változók esetében már a hagyományos C-ben megszoktuk, hogy a definíció során kezdeti értékeket is megadhatunk. A C++ filozófia a kezdeti értékadást kiemelt fontossággal kezeli (lásd például a *konstruktorok*at 3.7-nél), így szinte természetes, hogy ezt a lehetőséget a függvényparaméterekre is kiterjeszti.

A C nyelv (és a C++ is) megengedi, hogy egy függvényt kevesebb aktuális paraméterrel aktivizáljunk, mint ahány paramétert a formális paraméterlistában deklaráltunk. Ez sokszor kellemes, jól kihasználható tulajdonság, sokszor azonban kellemetlen, nehezen észrevehető mellékhatásokat eredményező hibaforrás is lehet. Gondoljunk csak a jól ismert `printf` függvényre, melynek működése a C ezen tulajdonságán alapul: Ha a formátum-specifikáló sztringben több kiirandó adatot határozunk meg, mint ahányat aztán ténylegesen felsorolunk a `printf` aktuális paraméterlistájában, akkor mindig valami "szemét" kerül standard outputra. Ehhez hasonlóan határozatlan paraméterértékekkel dolgozhat akármilyen más függvény is, ha véletlenül hiányos aktuális paraméterlistával hívtuk meg. Az ilyen – esetleg fatális hibát okozó – szituációk elkerülését szolgálja az a lehetőség, hogy egy függvény deklarációjakor a formális paraméterlistán az egyes paraméterekhez alapértelmezés-szerű (*default*) kezdeti értékeket rendelhetünk. Tekintsünk erre egy példát:

```
double triangle(double a=0, double b=0, double gamma=90);
```

Ha a fenti deklaráció szerinti függvényt arra használjuk, hogy egy háromszög harmadik oldalának hosszát kiszámítsuk a koszinusz-tétel segítségével (úgy, hogy a két ismert oldal által közbezárt szöget fokokban kell megadnunk), akkor a `gamma` paraméter kihagyásával a Pithagorasz-tételre egyszerűsödhet a probléma. Tehát a `triangle(3,4,135)` függvényhíváskor az aktuális paraméterek értéke rendre 3, 4 és 135, a `triangle(3,4)` híváskor 3, 4 és 90, `triangle(3)` hatására 3, 0, 90 lesz, és végül a `triangle()` függvényhívás esetében teljesen az alapértelmezés szerinti 0, 0, 90 számhármas lesz érvényben.

3.1.4 C ++ kommentek

Figyeljük meg a 3.1.2-es rész példaprogramjában, hogy a C++-ban két *slash* (/) karakterrel egysoros kommentárokat írhatunk. Ez igaz minden C++ rendszerre. A BORLAND C++ esetében ezt a kommentezési stílust alkalmazhatjuk akkor is, ha csak egyszerű C fordítóként akarjuk használni a rendszert, mindazonáltal megjegyzendő, hogy az egysoros kommentárok ilyen jelölése általában nem portábilis (mert a régebbi C fordítók nem fogadják el //-t. Természetesen a `/* .. */` páros és a // alkalmazása vegyesen is lehetséges.

3.2 Az OOP alapjai

Az objektum-orientált programozás (röviden OOP) a természetes gondolkodást, cselekvést közelítő programozási mód, amely a programozási nyelvek

tervezésének természetes fejlődése következtében alakult ki. Az így létrejött nyelv sokkal struktúráltabb, sokkal modulárisabb és absztraktabb, mint egy hagyományos nyelv. Egy OOP nyelvet három fontos dolog jellemez. Ezek a következők:

- Az *egységbezárás* (*encapsulation*) azt takarja, hogy az adatstruktúrákat és az adott struktúrájú adatokat kezelő függvényeket (Smalltalk, illetve a TURBO Pascal 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 típusát a C++-ban *osztály*nak (*class*) nevezzük.

- Az *öröklés* (*inheritance*) azt jelenti, hogy adott, meglévő osztályokból levezetett újabb osztályok öröklik a definálásukhoz használt alaposztályok már létező adatstruktúráit és függvényeit. Ugyanakkor újabb tulajdonságokat is definálhatnak, vagy régieket újraértelmezhetnek. Így egy osztályhierarchiához jutunk.

- A *többrétűség* (*polymorphism*) alatt azt értjük, hogy egy adott tevékenység (metódus) azonosítója közös lehet egy adott osztályhierarchián belül, ugyanakkor a hierarchia minden egyes osztályában a tevékenységet végrehajtó függvény megvalósítása az adott osztályra nézve specifikus lehet. Az ún. *virtuális függvények* lehetővé teszik, hogy egy adott metódus konkrét végrehajtási módja csak a program futása során derüljön ki. Ugyancsak a többrétűség fogalomkörébe tartozik az ún. *overloading*, aminek egy sajátságos esete a C nyelv standard operátorainak átdefiniálása (*operator overloading*).

Ezek a tulajdonságok együtt azt eredményezik, hogy programkódjaink sokkal struktúráltabbá, könnyebben bővíthetővé, könnyebben karbantarthatóvá válnak, mintha hagyományos, nem OOP-technikával írnánk őket. Hogy a C++ előnyeit élvezhessük, kicsit módosítanunk kell a programozásról alkotott képünket. Ehhez segít hozzá a fenti három tulajdonság részletesebb tárgyalása.

3.3 Egységbezárás

A C++ egyik fontos tulajdonsága, hogy lehetőségünk van az adataink és az őket manipuláló programkód összeforrasztására, egy egységbe zárására, egy osztályba foglalására. (Ez az ún. *encapsulation*.) Például tegyük fel, hogy definálunk egy karakterkészletet (*font*-ot) leíró adatstruktúrát (például egy fejrészt és az egyes karakterek tulajdonságait leíró tömböt), amely

kellő információt tartalmaz ahhoz, hogy a karaktereket kirajzolhassuk a képernyőre, és vannak függvényeink, amelyekkel a karaktereinket megjeleníthetjük, átszínezhetjük és mozgathatjuk a képen.

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 manipuláló függvények között még nincs explicit összerendeltség, továbbá más programozók egy másik modulból direkt módon is hozzáférhetnek az adatainkhoz, anélkül, hogy az adatok kezelésére szolgáló függvényeinket használnák. Ilyen esetekben az alap-adatstruktúra megváltozása fatális hibát okozhat egy nagyobb project-ben.

A C++-ban speciális tárolási egységfajták, az osztályok szolgálnak arra, hogy adatokat és a hozzájuk rendelt függvényeket egy egységként, egy *objektumként* kezelhessünk. Az osztályok alaptípusa a C++-ban a `class`, amely hasonlít a hagyományos C-ből ismert `struct`-ra és `union`-ra. A C++-ban a `struct` és `union` is egy bizonyos fajta osztályok. A `union`-ra a hagyományos értelemben továbbra is szükség van, a `struct`-ot a kulcsszót pedig a C++-ban alapvetően a hagyományos C-vel való kompatibilitás biztosítása végett tartották meg.

Tehát a fenti kulcsszavak szolgálnak arra, hogy osztályokat definiálhassunk. A C++-ban egy osztály típusú tárolási egység (`class`, vagy `struct`) függvényeket (ún. *függvénymezőket* – angolul *member functions*) kombinál adatokkal (*adatmezőkkel* – angolul *data members*), és az így létrejött kombinációt elrejtjük, elzárjuk a külvilág elől. Ezt értjük az *egységbezárás* alatt. Egy `class`-deklaráció hasonló a jól ismert struktúradeklarációhoz:

```
class font {
            ...
          };
```

A fenti deklaráció után a `font` típussal tetszőleges változó, objektumpéldány (*instance*) definiálható:

```
font times[10];
font *f_ptr;
```

Egy másik változás a hagyományos C-hez képest az, hogy egy `struct` vagy `class` típuscímke (fenti példánkban a `font`) kulcsszó elhagyásával önmagában is típusértékű, tehát közvetlenül is használható változók definiálására, deklarálására.

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 C és a C++ `union`-jai többé-kevésbé hasonlóak, egy C++ `union`-ban minden mező `public`, azaz mindenhonnan hozzáférhető, és ez a hozzáférési szint nem is változtatható meg. Egy `struct`-tal definiált osztályban alapértelmezés szerint minden mező `public` (ennek a hagyományos C-vel való kompatibilitás az oka, egyébként a `struct` osztályként való használata kerülendő), de ez megváltoztatható, míg egy `class`-szal definiált osztályban a mezők alapértelmezés szerint `private` hozzáférésűek, bár ez is módosítható. Az OOP-re jellemző az, hogy az adatmezőket privátnak, a függvénymezőket pedig publikusnak deklaráljuk. A továbbiakban csak a `class` típusú osztályokkal foglalkozunk, és javasoljuk, hogy a C++-ban programozók is csak a `class` kulcsszóval definiáljanak osztályokat.

Előbbi pédánknál maradva, a `font` típust `class`-ként (osztályként) deklarálva elérhetjük, hogy a privát (`private`) hozzáférésű karakter-leíró információt csak a hozzájuk rendelt publikus (`public`) függvényekkel lehessen kezelni. Így, ha a karakterfont leírásában valamit változtatunk, azt az adott projectben résztvevő többi programozó észre sem fogja venni (feltéve persze, hogy az adatmezőkkel együtt az azokat kezelő függvénymezőket is módosítjuk). A font-definíciónk többé már nem egy konkrét reprezentációhoz kötődik, hanem azokhoz a műveletekhez, amelyeket az egyes karakterekkel végezhetünk.

3.4 Öröklés

Mi jut eszünkbe az öröklés szóról? Talán az, hogy a gyermek milyen tulajdonságokat örökölt szüleitől, vagy gondolhatunk egy családfára is. Ugyanilyen családfa építhető fel osztályokkal is. A C++-ban deklarálhatók származtatott osztályok (*derived classes*), amelyek öröklik az őstípusok, az alaposztályok (*base classes*) adat- és függvénymezőit. Ez lehetővé teszi egy hierarchikus osztálystruktúra létrehozását.

Nézzünk meg erre egy egyszerű példát. Ha grafikus programot készítünk, szükségünk lehet arra, hogy kezeljük a grafikus képernyőn egy pont (*pixel*) helyének `x` és `y` koordinátáit. Hozzunk ezért létre egy `location` (hely) nevű struktúrát:

```
struct location {
                    int x;
                    int y;
                };
```

Tegyük fel, hogy szükségünk van az **x**, **y** koordinátában lévő pont tulajdonságára is, például nyilván akarjuk tartani, hogy egy kérdéses pont milyen szinű. Ezt megoldhatjuk a **color** (szín) felsorolt típusú változó bevezetésével. Így definiáljuk a **point** (pont) típust:

```
enum   colortype { black, red,    blue,
                   green, yellow, white };
struct point {
                int        x;
                int        y;
                enum colortype color;
              };
```

Ezek után deklarálhatunk **point** típusú változókat:

```
struct point origin, center,
             current_pos, new_pos;
```

A C++-ban a fenti deklarációt rövidebben is írhatjuk – lévén egy osztálydefinicónál megadott típuscímke önmagában is típus értékű:

```
point origin, center,
      current_pos, new_pos;
```

Mint láttuk, a **point** típusban is felhasználtunk olyan adatmezőket, amilyenek a **location**-ban szerepeltek. Jó lenne tehát, ha **point** definiálásához felhasználnánk a **location** típust. Nos, hagyományos C-ben a **point**-ot a **location** struktúra felhasználásával így definiálhatnánk:

```
struct point {
                struct location position;
                colortype       color;
              };
```

Ez nem más, mint struktúrák egymásbaágyazása (*nested structures*). Ugyanakkor a C++ esetében a deklarációban jelezhetjük, hogy a pont is egy hely. Ezt úgy tehetjük meg, hogy mind a **point** típust, mind a **location** típust **class**-ként definiáljuk, és **point**-ot a **location**-ból *származtatjuk*:

```
class location
        {
          protected:
              int x;
              int y;
        }
```

```
class point : location
              {
                    colortype color;
              };
```

Ez a deklaráció sokkal jobban kifejezi azt, hogy miben különbözik a `point` típus a `location` típustól. (A `protected` kulcsszó jelentését később tárgyaljuk majd.) A különbség teljesen nyilvánvalóan abban van, hogy a `point` egy hely (`location`), amelyhez valamilyen színt is rendeltünk.

A képernyőn a pont helyzete a `point` definíciójában explicit módon nem szerepel, mert azt a `location` típustól örökölte. Mivel a `point` típus a `location` leszármazottja, annak minden mezőjét örökli, de ugyanakkor a színt leíró mezővel bővítettük. Egy származtatott típus definíciós blokkjában csak az új, vagy megváltozott tulajdonságoknak megfelelő mezők szerepelnek. Természetesen a `point` típusból újabb osztályokat származtathatunk úgy, hogy tovább bővítjük tetszőleges típusú mezőkkel. Az így létrehozott típusok természetesen (az új tulajdonságaik mellett) hordozni fogják a `point` típus minden meglévő tulajdonságát, és használhatók lesznek minden olyan esetben, amikor `point` is használható.

Azt az eljárást, amely által az egyik típus örökli a másik típus tulajdonságait, öröklésnek hívják. Az öröklő a leszármazott típus (*descendant type*). Az őstípus (*ancestor type*) az, amelytől a leszármazott típus örököl.

Fenti példánkban a `location` az őstípus, a `point` pedig a leszármazott típus. Később látni fogjuk, hogy ez az eljárás korlátlanul folytatható. Definiálhatjuk a `point` leszármazott típusát, majd ennek a `point` leszármazott típusának a leszármazottját. A programtervezés legnagyobb része az OOP-ben a megfelelő osztály-hierarchiának, azaz az osztályok családfájának a létrehozását jelenti.

A `point` típus a `location` típusnak egy közvetlen leszármazottja (*immediate descendants*). Fordítva, a `location` típus a `point` típusnak közvetlen őstípusa (*immediate ancestor*). Egy objektumtípusnak bármennyi közvetlen leszármazottja lehet.

Az öröklés szintaktikailag a következőképpen nyilvánul meg. Tegyük fel, hogy az `alfa` azonosítójú, `point` típusú objektum `x` koordinátájára szeretnénk hivatkozni. Ezt a `point` típus hagyományos C szerinti beágyazott struktúrákkal történő defincíója esetén az `alfa.position.x` módon tehetnénk meg, míg C++-ban, mivel a `point` típus mindent örököl a `location` típustól, az `alfa.x` kifejezés a megfelelő hivatkozás.

Mint ahogy a C-ben egy típussal, a C++-ban a `point` osztály segítségével tetszőleges tárolási egységeket (újabb osztályokat, változókat, tömböket, osztályokra mutató pointereket, stb.) definiálhatunk, így jutva különböző objektum-példányokhoz:

```
point origin, line[80],
point *point_ptr1 = &origin;
point *point_ptr2 = line;
```

A C++-ban egy származtatott típus nemcsak egy őstípustól örökölhet (egy családfán pl. minden egyednek van apja is és anyja is). Azt, amikor egy származtatott típus több őstípusból került levezetésre, *többszörös öröklésnek* (*multiple inheritance*) nevezzük. A többszörös öröklés csak az AT&T 2.0-ás verziójú C++ fordítójával kompatibilis C++ rendszerekben lehetséges. Többszörös öröklés esetén az őstípusokat egymástól vesszővel elválasztva kell felsorolnunk. Ha tehát `father` és `mother` egy-egy létező osztály (például `class`-ként definiált típus), akkor segítségükkel a `child` típust a következőképpen deklarálhatjuk:

```
class child : father, mother
      {
       // Itt jonnek a 'child'-ra jellemzo
       // ujabb tulajdonsagok mezoi.
      };
```

A `child` tehát örökli a `father` és a `mother` minden tulajdonságát (azaz minden adat- és függvénymezőjét), és egyben újabb tulajdonságokkal (mezőkkel) rendelkezhet.

A mezőhozzáférés kapcsán a 3.8-as szakaszban látni fogjuk, hogy egy származtatott típusban az őstípusoktól örökölt függvénymezők hozzáférési szintjei módosíthatók az őstípus-listában a típusazonosítók előtt elhelyezett megfelelő kulcsszavakkal (`private`, `protected`).

Mivel a származtatott típusokkal mindent olyan művelet elvégezhető, ami a definiciója során felhasznált őstípusokhoz felhasználható, ezért egy származtatott típus mindig felülről kompatibilis az őstípusaival.

Összefoglalva tehát a fent leírtakat: Hagyományos C-ben egy struktúra nem tud örökölni, a C++ azonban támogatja az öröklést. A nyelvnek ez a kiterjesztése egy új tárolási egység-fajtában nyilvánul meg, amely hasonló a C-ben megszokott tárterületfoglaló tárolási egységekhez, a struktúrákhoz és a -unionokhoz, de azoknál sokkal hatékonyabb. A tárolási egységek ezen új fajtája az osztály. Hatékonysága abban áll, hogy míg a hagyományos C-ben hierarchikusan egymásbaágyazott struktúrarendszerben csak adatmezők "öröklődnek", addig a C++ osztályok esetében egy objektum egyéb tulajdonságai – nevezetesen az, hogy az adatmezőkkel mit és hogyan lehet csinálni – is (azaz a függvénymezők is) öröklődnek. Megjegyzendő, hogy a hagyomános C egymásba skatulyázott struktúrái esetében nincs szó igazi öröklésről, és a skatulyázás révén definiált újabb struktúrák, és a definiáláshoz felhasznált struktúrák között semmiféle típuskompatibiltás nincs.

A C++ osztályok típus-kompatibilitását a `point` – `location` páros felhasználásával az alábbiakban szemléltethetjük: egy `point` osztályba tartozó változó minden olyan esetben alkalmazható, amikor egy `location` osztályba tartozó érteket várunk. Tehát a

```
location *a;
point     b;
```

deklaráció mellett C++-ban az `a = &b` értékadás helyes, míg hagyományos C-ben, ha `point` és `location` egy-egy struktúra, akkor a fenti értékadás helytelen.

3.5 Többrétűség (polimorfizmus)

A *többrétűség* (vagy sokalakúság, sokoldalúság) a C++-ban azt jelenti, hogy egy adott őstípusból származtatott további típusok természetesen öröklik az őstípus minden mezőjét, így a függvénymezőket is. De az evolúció során a tulajdonságok egyre módosulnak, azaz például egy öröklött függvénymező 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 virtuális függvények biztosítják, hogy egy adott osztály-hierarchiában (származási fán) egy adott függvény különböző verziói létezhessenek úgy, hogy csak a kész program futása során derül, hogy ezek közül éppen melyiket kell végrehajtásra meghívni. Ezt a mechanizmust, azaz a hívó és a hívott függvény futási idő alatt történő összerendelését *késői összerendelésnek* (*late binding*) nevezzük. Erről majd a 3.9-es részben olvashatunk részletesebben.

A többrétűségnek a fordítási időben megvalósítható formája az ún. *overloading*. (A szerintünk is alkalmas magyar kifejezés helyett az eredeti angol kifejezést használjuk könyvünkben. Úgy gondoljuk, hogy a szószerinti fordítás, a *túlterhelés* nem fejezi ki e fogalom lényegét. A overloading helyett esetleg használhatnánk az átdefiniálás kifejezést, de mint később látni fogjuk, ez sem precíz megfogalmazás.) Nos, lássuk, miről is van szó:

A hagyományos C-ben egy adott névvel csak egy függvényt definiálhatunk. Például ha deklaráljuk az

```
int    cube(int number);
```

függvényt, akkor kiszámíthatjuk egy egész szám köbét, de elképzelhető, hogy egy `long`, vagy egy `double` szám köbére van szükségünk. Természetesen írhatnánk még két további függvényt erre a célra, de a `cube` azonosítót már nem használhatjuk:

```
long   lcube(long   lnumber);
double dcube(double dnumber);
```

A C++-ban mégis lehetőség van arra, hogy ugyanolyan azonosítóval lássuk el mind a három függvényt. Ezt hívják angolul *overloading*-nak. Ez tehát azt jelenti, hogy lehetőségünk van arra, hogy egy azonos névvel több különböző függvényünk legyen, melyek természetesen különböző adattípusokat dolgoznak fel:

```
int    cube(int    inumber);
long   cube(long   lnumber);
double cube(double dnumber);
```

Ha az azonos nevű függvények paraméterlistája különböző, a C++ gondoskodik arról, hogy egy adott típusú aktuális paraméterrel mindig a megfelelő függvényváltozatot hívja meg. Így például a `cube(10)` hivatkozás a `cube` függvény egész értéket visszaadó változatát hívja meg, míg `cube(2.5)` hívással a duplapontosságú lebegőpontos változatot aktivizáljuk. Felhívjuk a figyelmet arra a veszélyre, hogy igen könnyen a szándékainktól eltérő függvényváltozatot is aktivizálhatjuk. Például ha a `double` változatot akartuk meghívni a `2.0` értékkel, de véletlenül csak `2`-t írtunk, akkor az `int` verzió hívását tételezi fel a C++ fordító.

Az *overloading* általánosabban azt jelenti, hogy egy adatokon valamilyen operáció végrehajtását jelentő szimbólumhoz (függvényazonosítóhoz, operátorhoz) több, különböző típusra vonatkozó különböző jelentést rendelünk. Hogy egy adott adathoz egy művelet milyen értelmezése tartozik, az az adott adat típusától, vagy precízebben fogalmazva az adott operációt végrehajtó függvény ún. *paraméter-szignatúrájától* függ. Paraméter-szignatúrán azt értjük, hogy egy függvénynek mennyi és milyen típusú paramétere van, és az egyes paraméterek milyen sorrendben követik egymást. Tehát attól függően, hogy egy függvényt milyen típusú paraméterekkel aktivizálunk, mindig az aktuális paraméterekkel megegyező szignatúrájú függvényváltozat hívásának megfelelő kódot generál a C++ fordító. Ezt *siganture matching*-nek nevezzük. (Itt válik nyilvánvalóvá a hagyományos C programokban a modern stílusú függvénydefiníciók, illetve függvénydeklarációk fontossága: ezek nélkül a fordító program nem tudja egyeztetni az aktuális paraméterlistát a formális paraméterlistával.)

Az eddig elmondottakból nyilvánvaló, hogy az overloading nem átdefiniálást jelent, hiszen amikor az overloading által egy szimbólumhoz – a jelen példánkban a `cube` függvényazonosítóhoz – egy újabb jelentést rendeltünk, a korábbi jelentések nem vesztek el.

3.6 Függvénymezők definiálása

Ebben az alfejezetben az öröklés fogalmánál használt grafikus példát folytatjuk.

Az előzőekben láthattuk, hogy egy osztálynak nemcsak adatmezői, hanem függvénymezői (*function members*) is lehetnek. Egy függvénymező nem más, mint egy, a class definíción belül deklarált függvény. (A függvénymezőket más OOP nyelvekben, például a Smalltalk-ban, vagy a Turbo Pascal-ban metódusoknak hívják).

Most deklaráljunk egy `get_x` azonosítójú függvénymezőt a `point` osztály számára. Ezt kétféleképpen tehetjük meg:

a) vagy az osztály-definíción belül definájuk a függvényt (ún. *implicit inline* függvényként),

b) vagy az osztálydefiníción belül csak deklaráljuk, és azután valahol később definiáljuk.

A két módszer különböző szintaxissal rendelkezik. Lássunk egy példát az első lehetőségre:

```
class point
      { int       x;
        int       y;
        colortype color;
        int       get_x(void) { return x; };
        ....
        // implicit inline definicio
      };
```

Ez a fajta definíció alapértelmezésben inline függvényként definiálja a `get_x`-et. Ez az ún. implicit inline függvény definició. Figyeljük meg, hogy az implicit inline függvénydefiníció szintaxisa a hagyományos C-ben megszokottak szerinti.

A második lehetőségnél a C-szintaxissal csak deklaráljuk a `get_x` függvényt a `point` osztályban, és később adjuk meg a függvény teljes definicióját:

```
class point
      { int       x;
        int       y;
        colortype color;
        int       get_x(void);
        // majd definialjuk
```

```
        ....
      };
int point::get_x(void)
{
   return x;
}
// Itt jobb lenne az implicit inline definicio
```

Figyeljük meg alaposan, hogy használtuk a :: hatáskört definiáló operátort (*scope resolution operator*) a `point::get_x` függvénydefinícióban. A `point` osztályazonosító szükséges ahhoz, hogy a fordítóprogram tudja, melyik osztályhoz is tartozik a `get_x` függvénydefiníciója, valamint tudja azt is, hogy mi az érvényességi tartománya. Ez tehát azt jelenti, hogy a `pont::get_x` függvény (a hagyományos globális változókon kívül) csak a `point` osztályhoz tartozó objektumok mezőihez férhet hozzá. Természetesen létezhet több, más osztályhoz tartozó `get_x` függvény is. (Erre a lehetőségre – bár nem explicit módon – utaltunk a többrétűség tárgyalásánál.) Az osztály-deklaráción belüli függvény-definició esetében természetesen nem volt szükség az érvényességi tartományt definiáló `point::` előtagra, hiszen a `get_x` függvény hovatartozása abban az esetben teljesen egyértelmű volt.

A hovatartozás definiálásán túlmenően a `point::`-nak más szerepe is van. Hatása kiterjed az őt követő függvénydefinícióra oly módon, hogy a `get_x` függvényen belül az `x` azonosítójú változóra való hivatkozás a `point` struktúra `x` azonosítójú adatmezőjére vonatkozik, valamint a `get_x` függvény a `point` osztály hatáskörébe kerül (lásd a 3.6.2-es részt).

Függetlenül attól, hogy a függvénymezőt milyen módon definiáljuk, a lényeg az, hogy van egy `get_x` nevű függvényünk, ami szorosan hozzá van kapcsolva a `point` névvel ellátott osztályhoz. Mivel a `get_x`-et mezőként definiáltuk a `point` számára, a `point` minden adatmezőjéhez hozzáférhet. Ebben az egyszerű példánkban `get_x` egyszerűen csak `x` értékét szolgáltatja.

Elképzelhető, hogy egy származtatott típusban lokálisan deklarálunk egy függvénymezőt, melynek azonosítója és paraméterlistája is megegyezik egy őstípusban definiált függvénymező azonosítójával és paraméterlistájával, ugyanakkor a definíció során fel szeretnénk használni az őstípusbeli változatot. Mivel a későbbi definícióban lévő függvénymező a saját hatáskörében elfedi a korábbi definíciót, a származtatott típusban csak az érvényességi tartományt definiáló operátor segítségével használhatjuk az őstípusban definiált, "elfedett" függvénymezőt. Erre vonatkozóan (az eddigi példánkat egy kicsit félretéve) tekintsük az alábbi deklarációkat:

```
class rectangle { int   a, b;   // Negyszog adatai
                  ...
                  void  show(void); // Kirajzolja
```

```
                };
class triangle  { int   a, b, c;// Haromszog adatai
                  ...
                  void  show(void); // Kirajzolja
                };
class house : rectangle,          // Ostipusok
              triangle
                 {                // Egy hazat definial
                   ......         // Tetszoleges mezok
                   void show(void); // Kirajzolja
                 };
// Az egyes fuggvenymezok definicioi:
void rectangle::show( ) { ... } // Egy negyszoget rajzol
void triangle::show( )  { ... } // Egy haromszoget rajzol
void house::show( )             // Egy hazat rajzol
{
  // Itt az  egyszeru  'show'  azonosito  a  'house'  tipus
  // fuggvenymezojet jelenti, a 'show' hivas rekurzio lenne

  rectangle::show( ); // alap
  ...
  triangle::show( );  // teto
  ...
  ::show( ); // Egy globalis, 'show' fuggveny hivasa
};
```

Amint az fenti példából is látszik, ha egy hagyományos, nem egy osztály függvénymezőjeként deklarált **show** függvényt akarunk meghívni, akkor a függvénynév elé egy "üres" érvényességi tartományt definiáló operátort (::) kell tennünk (azaz nem utaljuk egy osztály hatáskörébe sem, így a teljes programra nézve globális lesz a függvény). Ezután a kitérő után a foglalkozzunk tovább a megszokott **point** / **location** példával.

3.6.1 Függvénymezők aktivizálása

A függvénymezők egy adott típusú adathalmazon végrehajtandó műveleteket jelentenek. Amikor tehát a **get_x** függvényt meghívjuk, tudatnunk kellene vele azt is, hogy most éppen melyik definiált **point** típusú objektum-példány (*object instance*) **x** koordinátáira van szükségünk, éppen ezért valahogy ezt az információt is közölnünk kell a fordítóprogrammal. A megoldás a hagyományos C struktúra-mezőkhöz való hozzáférés szintaktikájának kiterjesztése. Ha **origin** egy **point** típusú objektum-példány, akkor

az `origin.x` hivatkozáshoz hasonlóan az

```
origin.get_x( )
```

kifejezés az `origin` objektum-példány `get_x` függvénymezője által szolgáltatott értéket jelenti. A hagyományos C struktúra adatmező-hozzáféréshez hasonlóan itt is a '.' (pont) karakter játssza az osztály mezőkiválasztó operátorának (*class component selector*) szerepét. Az általános szintaxis a következő:

objektumnév.függvénymező-név(argumentumlista)

Az előbbiekhez hasonlóan, ha `p_ptr` egy `point*` típusú mutató, akkor a

```
p_ptr->get_x( )
```

kifejezés a `p_ptr` által megcímzett `point` típusú objektum `get_x` függvénymezője által szolgáltatott értéket adja.

3.6.2 A `this` nevű, implicit mutató

Egy függvénymezőben direkt módon is hivatkozhatunk arra az objektumra, amelyiknek a függvénymezejét aktivizáltuk. Például a

```
class cc {
            int m;
            public:
            ....
            int read_m(void) { return m; }
          };
```

deklaráció esetén, ha `aa` és `bb` `cc` típusúak, akkor az `aa.read_m( )`, illetve a `bb.read_m( )` függvényhívások esetén rendre az `aa.m`, illetve `bb.m` értékeket kapjuk. A függvényhívás során úgy derül ki, hogy melyik objektum adatmezőit kell használni, hogy minden függvénymező számára implicit módon deklarálásra kerül egy `this` nevű pointer. Ha tehát a `read_m` függvény egy `cc` típusú osztály függvénymezője, akkor a `read_m`-en belül `this` egy `cc*` típusú pointer, ilyen módon az első `read_m` hivatkozásnál `this` az `aa` változóra, míg a második hivatkozás alkalmával a `bb`-re mutat, azaz az első esetben `this == &aa`, a második esetben pedig `this == &bb`.

A `this` mutató explicit módon is megjelenhet a függvénymezők definíciója során:

```
class cc {
```

```
    int m;
    public:
    int read_m(void) { return this->m; }
};
```

3.7 Konstruktorok és destruktorok

Két speciális, előredefiniált függvénymező-fajta létezik, amelyek kulcszerepet játszanak a C++-ban. Ezek a *konstruktorok* (*constructors*) és a *destruktorok* (*destructors*). Hogy jelentőségüket megértsük, tegyünk egy kis kitérőt.

Általános probléma a hagyományos nyelveknél az *inicializálás*. Mielőtt használnánk egy adatstruktúrát, gondoskodnunk kell arról, hogy megfelelő tárterületet biztosítsunk az adatstruktúra számára és megfelelő kezdeti értékkel rendelkezzen a adott típusú változó. Tekintsük a `point` osztály egy korábbi deklarációját:

```
class point {
        int       x;
        int       y;
        colortype color;
};
```

Kezdő programozók az egyes adatmezőket így inicializálnák:

```
point a_point;
....
a_point.x = 10;
a_point.y = 25;
a_point.colortype = black;
```

Ez a kezdeti értékadás korrekt, de csak az `a_point` változónak ad értéket. Ha több `point` típusú változót akarunk inicializálni, akkor többször meg kell ismételni a fentihez hasonló értékadó utasításokat. A következő természetes lépés az lenne, hogy írunk egy inicializáló függvényt, amely tetszőleges, argumentumként átadott `point` típusú objektumot inicializál.

```
void init_point(point* target, int newx, int newy)
{
    target->x = newx;
    target->y = newy;
    target->color = black;
}
```

Ezt a függvényt jól terveztük meg: tetszőleges `point` típusú objektumot inicializál. De miért kell megadnunk az inicializáláshoz az osztály típusát és azt a konkrét objektumot, amelynek kezdeti értéket kell adnunk? A válasz egyszerű: azért, mert nem függvénymezőként deklaráltuk az inicializáló függvényt. Amire igazán szükségünk van, az egy olyan függvénymező, amelyik tetszőleges `point` típusú objektumot automatikusan inicializál. Az ilyen függvénymezőt *konstruktornak* nevezzük.

A C++ filozófiájának egyik lényeges eleme, hogy a felhasználó által definiált adattípusok ugyanolyan integráns részei legyenek egy programnak, mintha beépített típusok lennének. Ezért a C++ különleges függvénymezőket kínál számunkra, amelyek meghatározzák, hogy milyen módon hozhatók létre egy adott osztály objektumai. Ezek a konstruktorok. Egy konstruktor lehet a felhasználó által definiált, vagy lehet a C++ rendszer alapértelmezése szerinti. A konstruktorokat akár explicit, akár implicit módon meghívhatjuk. A C++ mindig automatikusan meghívja a megfelelő konstruktort, amikor egy adott osztállyal objektum-példányt definiálunk. Ez akár definició, akár egy objektum másolása, vagy akár a `new` operátorral (lásd később) dinamikusan létrehozott objektumok esetében megtörténik.

A *destruktorok* – mint ahogy nevük is mutatja – egy konstruktor által korábban létrehozott objektum megszüntetésében segítenek: törlik a mezők tartalmát és általában felszabadítják az objektum által elfoglalt tárterületet. A destruktorok szintén explicit módon (lásd a `delete` operátort később), akár implicit módon – például egy automatikus változó megszűnésekor – aktivizálhatók. Ha nem definiálunk mi magunk destruktort egy adott osztályhoz, a C++ a saját alapértelmezése szerinti változatát fogja használni. Egy általunk definiált destruktor egy objektum megszűnésével kapcsolatos kiegészítő műveleteket végezhet, például adatokat írhat diszkre, file-okat zárhat le, képernyőt törölhet, stb.

3.7.1 Konstruktorok definiálása

Most tekintsük át először a konstruktorok definiálását a megszokott `point` típusunk esetében; bővítsük ki az eddigi deklarációt egy konstruktorral.

```
class point
      {
        int       x;
        int       y;
        colortype color;
        int       get_x(void) { return x; }
        int       get_y(void) { return y; }
        colortype get_color(void);
```

```
        void      show(void);
        void      hide(void);
        point(int newx, int newy);
                  // konstruktor deklaracio
    };
point::point(int newx = 0, int newy = 0)
{                 // konstruktor definicio
  x = newx;
  y = newy;
  color = black;
}
```

A konstruktort ugyanúgy deklarálhatjuk, mint minden egyéb függvénymezőt. Figyeljük ugyanakkor meg, hogy a konstruktor neve és az osztály neve ugyanaz, mind a kettő `point`. Innen tudja a fordítóprogram, hogy az adott függvénymező 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 sem adhat vissza.) A fenti konstruktor-deklarációt figyelembe véve az alábbi módon deklarálhatunk egy `point` típusú objektumot:

```
point a_point(100,200);
```

Ez a deklaráció aktivizálja a fent definiált konstruktort, és ennek következtében a pont `x` és `y` koordinátája rendre 100 ill. 200 lesz. Az *overloading* a konstruktorok esetében is alkalmazható, így egy osztálynak több konstruktora is lehet, és mindig az aktuális paraméterlista alapján dől el, hogy melyik változatot kell aktivizálni. Ha nem definiálunk mi magunk egy konstruktort, akkor a C++ generál egyet, amelynek nincsenek argumentumai. Egy további C++ trükk, hogy az általunk definiált konstruktor argumentumaihoz egy alapértelmezést rendelhetünk, mint ahogy azt a `point::point` függvény esetében is tettük. Ekkor a

```
point a_point(5);
```

definíció hatására az `x` koordináta értéke 5 lesz, míg az `y` koordináta értéke az általunk adott alapértelmezés szerinti 0 marad.

Egy speciális konstruktor az ún. *másoló konstruktor* (*copy constructor*). Ha `s` egy osztály, akkor a hozzá tartozó másoló konstruktor alakja `s::s(s&)`. A másoló konstruktor jelentőségét megérthetjük, ha arra gondolunk, hogy sztringek tárolására hozunk létre egy `s` osztályt (tehát nem

elégszünk meg a C karaktertömbjeivel, hanem például olyen kifinomult sztringkezelő rendszert szeretnénk kiépíteni, mint amilyen miatt sokan még mindig ragaszkodnak a BASIC-hez). Ekkor ha `a` és `b` egy-egy `s` osztályba tartozó sztringváltozó, és elvégezzük az `a = b` értékadást, akkor nem szeretnénk azt, hogy `b` tartalmának megváltoztatása maga után vonja `a` tartalmának megváltozását. Hagyományos, egyszerű C sztringkezelés mellett a fenti értékadás csak azt eredményezné, hogy egy karakter-pointer értékét egy másik pointer változóba másoljuk, így ugyanazt a karaktertömböt címzik meg, így a "másolat" nem védhető meg az "eredeti" sztringben végrehajtott módosításoktól.

3.7.2 Destruktorok definiálása

Ahogy konstruktorokat definiálhatunk, ugyanúgy definálhatunk saját magunk destruktorokat is.

A statikus objektumok számára a C++ futtató rendszer a `main` meghívása előtt foglal tárterületet és a `main` befejezése után felszabadítja azt. Az `auto` objektumok esetében a tárterületfelszabadítás akkor történik meg, amikor az adott változó érvényét veszti (tehát az objektum definiciót tartalmazó blokk végén). Az általunk definiált destruktorokat akkor aktivizálja a C++ futtató rendszer, amikor egy objektumot meg kell szüntetni. A destruktor definíció hasonlít a konstruktor definícióra. Ha a `point` típusú objektumoknak `point::point` a konstruktora, akkor `point::~point` lesz a destruktoruk. A formai különbséget csak a ~ (tilde) karakter jelenti (ami a komplementálás operátora).

```
class point
    {
      int       x;
      int       y;
      colortype color;
      int       get_x(void) { return x; }
      int       get_y(void) { return y; }
      colortype get_color(void);
      void      show(void);
      void      hide(void);
      point(int newx, int newy); // konstrktor
      ~point(void);              // destruktor
    };
```

A fenti példa első ránézésre felesleges, hiszen elég lenne a C++ futtató rendszer alapértelmezés szerinti destruktorára hagyatkozni. Igen ám, de

egy megszűnésre ítélt pontot a képernyőről is el kell tüntetni. Ez azt jelenti tehát, hogy egy olyan destruktor függvényt kell definiálnunk a `point` típus számára, amelyik szükség esetén (például ha a `hatterszin == get_color()` igaz) a `hide` függvény meghívásával az adott pontot le is törli a képernyőről.

3.8 Mezőhozzáférés

Ahogy a 3.1-es szakaszban az egységbezárás kapcsán már utaltunk rá, egy osztály egyes mezőihez való hozzáférést mi magunk is kézben tarthatjuk a C++ által nyújtott alapértelmezés felülbírálásával.

A C++-ban a `struct` típusú osztályokban csak részben érvényesül az egységbezárás. Nevezetesen az *egység* megvan, ugyanis az adatmezők és a függvénymezők egybe vannak forrasztva, de nincsenek *elzárva* a külvilágtól. Ennek oka az, hogy az alapértelmezés szerint egy `struct` minden mezője publikus, a külvilágból szabadon hozzáférhető. A jó C++ programtervezés szem előtt tartja az adat- vagy még inkább az *információrejtést*, azaz egy osztály egyes mezőit privátnak, vagy védettnek deklarálja, és ezzel egyidejűleg jól definiált, körülhatárolt "*hivatalos*" külső hozzáférést biztosít hozzájuk. Általános elvként tekinthetjük azt, hogy az adatmezők privátak vagy védettek és a csak belső műveleteket végrehajtó függvénymezők szintén privátak, míg a külvilággal való kapcsolattartást publikus függvénymezők biztosítják. Ezért javasoljuk a `class` használatát a `struct` helyett, hiszen a `class` alapértelmezés szerinti mezőhozzáférési szintjei pontosan a fenti kívánalmaknak felenek meg.

A C++-ban három, a cimkékhez formailag hasonló módon használható kulcsszó áll rendelkezésünkre, hogy a mezőhozzáférést mi magunk állíthassuk be egy osztály-definíció során. A mezőhozzáférés-szabályozás általános szintaktikája a következő:

hozzáférési mód : *mezőlista*

ahol a *hozzáférési mód* a `public`, `private`, vagy `protected` kulcsszavak egyike lehet. Például ha `point`-ot `struct`-ként deklaráljuk, célszerű az alábbi mezőhozzáférést beállítani:

```
struct point {    // Ez most csak a pelda miatt struct !!!
            private:                    // Privat
                 int       x;
                 int       y;
                 colortype color;
            public:                     // Publikus
```

```
        int       get_x(void) { return x; }
        colortype get_color(void);
        point(int newx, int newy);
    };
```

Egy mezőhozzáférést módosító kulcsszó hatása egy következő hasonló kulcsszóig, vagy az őt tartalmazó blokkzáró kapcsos zárójelig tart.

3.8.1 Mezőhozzáférési szintek

Mint említettük, három különböző szinten érhetjük el egy osztály egyes mezőit. Most tekintsük át részletesen, mit is jelent e három szint.

A `private`-ként deklarált mezőket csak az ugyanabban az osztályban deklarált függvénymezők érhetik el. A `protected` mezőket az ugyanabban az osztályban, és az adott osztályból származtatott további osztályokban definiált függvénymezők érhetik el. A `public` mezők korlátozás nélkül elérhetők mindenhonnan, ahonnan az adott osztály is elérhető.

A `struct`-tal definiált osztályok mezői alapértelmezés szerint mind publikusak (`public` elérésüek), így ahhoz, hogy az igazi *egységbezárást* megvalósíthassuk, a külvilág elől elzárandó mezőket mi magunk kell, hogy a `private` kulcsszóval privátnak deklaráljuk (a `private` részt esetleg követő újabb nyilvános részeket megint a `public` kulcsszóval kell megnyitnunk a külvilág számára). Minthogy egy jó C++ programozó mindent privátnak tekint és csak azt mondja meg külön, hogy mely mezők publikusak, osztályok definiálására előnyösebb a `class`-t használni a `struct` helyett, ugyanis egy class minden mezője alapértelmezés szerint `private`. (Az egyetlen különbség a `struct` és a `class` között tehát a mezőhozzáférés alapértelmezésében van.) A `point` típust `class`-ként definiálva így célszerű a mezőhozzáférést a következők szerint szabályozni:

```
class point { // Alapertelmezes szerint privat:
        int       x;
        int       y;
        colortype color;
    public:                 // Publikus:
        int       get_x(void) { return x; }
        int       get_y(void) { return y; }

        colortype get_color(void);
        void      show(void);
        void      hide(void);

        point(int newx, int newy);
```

```
                ~point(void);

            };
```

Nincs szükségünk tehát a **private** kulcsszóra az adatmezők definiálásakor, ugyanakkor a függvénymezőket **public**-nak kell deklarálni, hogy létrehozhassunk **point** típusú objektumokat és hogy belőlük értékeket olvashassunk ki.

3.8.2 Mezőhozzáférés és öröklés

A mezőhozzáférés és az öröklés egymással szoros kapcsolatban állnak. Ennek áttekintéséhez a **point** típusnak a **location**-ból levezetett változatát használjuk úgy, hogy már a mezőhozzáférést is szabályozzuk.

```
enum  colortype { black, red, blue, green, yellow, white };
class location {
            protected:// A csak vedett mezoket a szar-
                      // maztatott tipusok is elerhetik
              int     x;
              int     y;
            public:   // Publikus fuggvenyek a kulvilagnak
              int     get_x(void);
              int     get_y(void);
              location(int ix, int iy); // Konstruktor
              ~location(void);          // Destruktor

          };
class point : public location
          { // A publikus szarmaztatas azt jelenti, hogy
            // 'location' mezoinek hozzaferesi szintjei
            // valtozatlanul oroklodnek.
            protected: // A leszarmazottak is lathatjak
              colortype color;
            public:
              colortype get_color(void);
              void      show(void);
              void      hide(void);

              point(int ix, iy);        // Konstruktor
              ~point(void);             // Destruktor

          };
```

A `location` adatmezőit – a `class`-ra vonatkozó `private` alapértelmezéssel szemben – csak védettnek deklaráljuk, hogy a származtatott típusok – például a `point` – is hozzáférhessenek, de azért akárhonnan mégse legyenek elérhetők. A függvénymezőket (a konstruktort is ideértve) nyilvánosnak deklaráltuk, hogy bárki létrehozhasson `location` típusú objektumot, illetve hogy ki lehessen olvasni a koordinátákat.

Egy származtatott típust általában a következőképpen deklarálhatunk:

`class` *D* : *hozzáférés-módosító B* { ... };

illetve

`struct` *D* : *hozzáférés-módosító B* { ... };

ahol *D* a származtatott (*derived*) osztály típusazonosítója, *B* pedig az alap osztály (*base class*) típusazonosítója. A *hozzáférés-módosító* vagy `public`, vagy `private` lehet; használata opcionális. Mint már említettük, `class` esetén a *hozzáférés-módosító* alapértelmezés szerint `private`, `struct` esetén pedig `public`. Egy származtatott osztály mezőihez való hozzáférést csak szigorítani lehet az alaptípus egyes mezőinek hozzáférési szintjeihez képest.

Egy osztályt egy másikból publikus vagy privát módon származtathatunk. Ha a *hozzáférés-módosító* a `private`, akkor az összes `protected` és `public` mező `private` lesz. Ha a *hozzáférés-módosító* a `public`, akkor a származtatott típus változás nélkül örökli az alaptípus mezőhozzáférési szintjeit. Egy származtatott típus az alaptípus minden mezőjét örökli, de azok közül csak a `public` és a `protected` mezőket használhatja korlátozás nélkül. Az alaptípus `private` mezői a származtatott típusban direkt módon nem érhetők el. A legutóbbi példában `location`-t és `point`-ot úgy deklaráltuk, hogy megfelelő információrejtés mellett további, összetettebb osztályokat deklarálhassunk.

A hozzáférés módosítót célszerű explicit módon megadni, függetlenül egy osztályfajtára (`class`-ra, vagy `struct`-ra) vonatkozó alapértelmezéstől.

3.9 Virtuális függvények

3.9.1 Késői összerendelés

Az előző részekben egy összetett grafikus programrendszer lehetséges alaptípusainak példáján mutattuk be az OOP egyes jellemzőit. A `point` típus

a `location`-ból lett származtatva, ugyanígy `point`-ból létrehozhatunk egy vonalakat leíró osztályt, amelyet például `line`-nak nevezhetünk, és mondjuk `line`-ból származtathatunk mindenféle poligont. A poligonokra két vázlatos példát mutattunk (`triangle`, `rectangle`), ezekből felépítve pedig már egy "valóságos" objektumok leírására való típus, a `house` vázlatos leírását adtuk. (Ezek a példák természetesen nem egy működő program létrehozására, hanem sokkal inkább egy-egy OOP fogalom megvilágítására voltak kihegyezve.) Az eddigi példák többségének ugyanakkor volt egy közös vonása, nevezetesen az, hogy deklaráltunk bennük egy-egy `show` azonosítójú függvénymezőt azzal a céllal, hogy az az adott típusú objektumot a képernyőn megjelenítse. Az alapvető különbség az általunk elképzelt objektumtípusok között az, hogy milyen módon kell őket a képernyőre kirajzolni. Az egésznek van egy nagy hátránya: akárhányszor egy újabb alakzatot leíró osztályt definiálunk, a hozzátartozó `show` függvényt mindannyiszor újra kell írnunk. Ennek az az oka, hogy a C++ alapvetően háromféleképpen tudja az azonos névvel ellátott függvényeket egymástól megkülönböztetni:

- a paraméterlisták különbözősége (az ún. paraméterszignatúra) alapján, tehát `show(int i, char c)` és `show(char *c, float f)` egymástól különböző függvények,
- az érvényességi tartományt definiáló operátor alapján, tehát a `rectangle::show` és a `triangle::show`, valamint a `::show` függvények különbözőek,
- illetve az egyes objektum-példányok függvénymezőit a mezőkiválasztó operátor(ok) segítségével.

Az ilyenfajta függvényhivatkozások feloldása mind fordítási időben történik. Ezt nevezzük *korai*, vagy *statikus összrendelésnek* (*early binding*, *static binding*).

Egy komolyabb grafikus rendszer esetében azonban igen gyakran előfordul az a helyzet, hogy csak az osztálydeklarációk állnak rendelkezésre forrásállományban (`.h` kiterjesztésű include file-okban), maguk a függvénymező definíciók pedig csak tárgykód formájában vannak meg (`.obj` file-okban). Ebben az esetben, ha a felhasználó a meglévő osztályokból akar újabbakat származtatni, akkor a korai kötés korlátai miatt nem lesz könnyű dolga az alakzatmegjelenítő rutinok megírásánál. A C++ ezt a problémát az ún. *késői kötés* (*late binding*) lehetőségével hidalja át. Ezt a késői kötést speciális függvénymezők, a *virtuális függvények* (*virtual functions*) teszik lehetővé.

Az alapkoncepció az, hogy a virtuális függvényhívásokat csak futási időben oldja fel a C++ rendszer – innen származik a késői kötés elnevezés. Tehát azt a döntést, hogy például melyik `show` függvényt is kell aktivizálni,

el lehet halasztani egészen addig a pillanatig, amikor a ténylegesen megjelenítendő objektum pontos típusa ismertté nem válik a program futása során. Egy virtuális **show** függvény, amely el van "rejtve" egy előzetesen lefordított modulkönyvtár egy *B* alaposztályában, nem lesz véglegesen hozzárendelve a *B* típusú objektumokhoz olyan módon, ahogy azt normál függvénymezők esetében láttuk. Nyugodtan származtathatunk egy *D* osztályt a *B*-ből, definiálhatjuk a *D* típusú objektumok megjelenítésére szolgáló **show** függvényt. Az új forrásprogramot (amelyik az új *D* osztály definícióját is tartalmazza) lefordítva kapunk egy `.obj` file-t, amelyet hozzászerkeszthetünk a grafikus könyvtárhoz (a `.lib` file-hoz). Ezután a **show** függvényhívások, függetlenül attól, hogy a *B* alaposztálybeli függvénymezőre, vagy az új, *D* osztálybeli függvénymezőre vonatkoznak, helyesen lesznek végrehajtva. Lássuk ezt az egész fogalomkört egy kevésbé absztrakt példán.

Tekintsük a jól bevált **point** típust, és egészítsük ki azt egy olyan függvénnyel, amelyik egy ilyen típusú objektum által reprezentált pontot a képernyő egyik helyéről áthelyezi egy másikra helyre. A **point**-ból azután származtassunk egy körök leírására alkalmas **circle** típust (olyan meggondolás alapján, hogy egy pont egy olyan kör, amelynek sugara 0, tehát a **point** típust egyetlen adatmezővel, a sugárral kell kiegészíteni).

```
enum   colortype { black, red, blue, green, yellow, white };
class location {
                  protected:
                    int  x;
                    int  y;
                  public:
                    location(int ix, int iy);
                    ~location(void);

                    int get_x(void);
                    int get_y(void);
                };
class point : public location
                {
                  protected:
                    colortype color;
                  public:
                    colortype get_color(void);
                    void      show(void);
                    void      hide(void);
                    void      move(int dx, int dy);
                    point(int ix,int iy);
```

```
                    ~point(void);

                };
 // A point:: fuggvenyek definicioja:
...
void point::move(int dx, int dy)
{   // dx, dy offsetekkel athelyezi
    if (color) hide( ); // Ez itt a point::hide
    x += dx;
    y += dy;
    if (color) show( ); // Ez itt a point::show
}
class circle : public point
                {
                  protected:
                    int       radius;       // A sugara
                  public: // Konstruktorhoz kezdeti x,y,r
                    circle(int ix,int iy,int ir);
                    colortype get_color(void);
                    void      show(void);
                    void      hide(void);
                    void      move(int dx, int dy);
                    void      zoom(int factor);
                };
// A circle:: fuggvenyek definicioja:
...
void circle::move(int dx, int dy)
{   // dx, dy offsetekkel athelyezi
    if (color) hide( ); // Ez itt a circle::hide
    x += dx;
    y += dy;
    if (color) show( ); // Ez itt a circle::show
}
...
```

Figyeljük meg, hogy a két **move** függvény mennyire egyforma! A visszatérési típusuk **void** és a paraméter-szignatúrájuk is azonos, sőt, még a függvénytörzsek is azonosnak tűnnek. Hát akkor mi alapján tudja a fordító, hogy mikor melyik függvényről van szó? Ez esetben – ahogy azt a normál függvénymezők esetében láttuk – a **move** függvényeink abban különböznek, hogy más-más osztályhoz tartoznak (**point::move**, illetve **circle::move**). Fölvetődik a kérdés: Ha a két **move** függvény törzse is egyforma, akkor miért

kell belőlük 2 példány, miért ne örökölhetné `circle` ezt a függvényt `point`-tól? Nos azért, mert csak felületes ránézésre egyforma a két függvénytörzs, ugyanis más-más `hide`, illetve `show` függvényeket hívnak – ezeknek csak a neve és a szignatúrája azonos. Ha `circle` a `point`-tól örökölné a `move` függvényt, akkor az nem a megfelelő `hide`, illetve `show` függvényeket hívná: nevezetesen a körre vonatkozó függvények helyett a pontra vonatkozókat aktivizálná. Ez azért van így, mert a korai kötés következtében a `point::hide`, illetve a `point::show` lenne a `point` típus `move` függvényéhez hozzárendelve, és az öröklés által a `circle` típus `move` függvénye is ezeket függvényeket hívná. Ezt a problémát úgy oldhatjuk meg, ha a `show`, illetve a `hide` függvényeket virtuális fügvényekként deklaráljuk. Ekkor nyugodtan írhatunk egy közös `move` függvényt a `point` és a `circle` számára (pontosabban fogalmazva a `circle` típus örökölhetné a `point` típus `move` függvényét), és majd futási időben fog kiderülni, hogy melyik `show`, illetve `hide` függvényt kell meghívni.

3.9.2 Virtuális függvények deklarálása

A virtuális függvények deklarálásának szintaktikája nagyon egyszerű: a függvénymező első deklarálásakor helyezzük el a `virtual` típusmódosító szót is:

```
virtual void show(void);
virtual void hide(void);
```

Figyelem! Csak függvénymezők deklarálhatók virtuális függvényekként. Ha egy függvényt egyszer már virtuálisnak deklaráltunk, akkor egyetlen származtatott osztályban sem deklarálhatjuk újra az adott függvényt ugyanolyan paraméter-szignatúrával, de más visszatérési típussal. Ha egy egyszer már virtuálisnak deklarált függvényt egy származtatott típusban újra deklaráljuk és az újbóli deklaráció alkalmával ugyanolyan viszszatérési típust és paraméter-szignatúrát alkalmazunk, mint a korábbi deklarációban, akkor az újonnan deklarált függvény automatikusan virtuális függvény lesz. Természetesen egy virtuális függvényt más paraméter-szignatúrával újradeklarálhatunk, de akkor erre a változatra a virtuális függvénykezelő rendszer nem fog működni. Az azonos szimbólumhoz rendelt különböző jelentések (*function overloading* (függvénynév átdefiniáló) – lásd a többrétűségnél) használata sok veszélyt rejthet, úgyhogy ezt csak gyakorlott C++ programozóknak ajánljuk.

3.10 Dinamikus objektumok

Eddigi példáinkban csak statikus objektumokkal volt dolgunk (amelyek számára a fordítóprogram foglal le tárterületet a fordítás során), jóllehet a hagyományos C-ben már megszokhattuk, hogy változóink egy része számára mi magunk foglalunk le tárterületet, és a változókat megszüntetve felszabadítjuk a memóriát, ha az adott változókra többé már nincs szükségünk.

Természetesen a C++-ban sem kell lemondanunk a dinamikus tárkezelésről az objektumok kapcsán, akár használhatjuk is a hagyományos C-ben megismert tárkezelő függvényeket, például a malloc-ot is. Ezt mégsem javasoljuk objektumok esetében, mert a C++ a hagyományos tárkezelő függvényeknél jóval hatékonyabb módszereket kínál. Például biztosítja, hogy a dinamikusan kezelt objektumok esetében is aktivizálódjanak a megfelelő konstruktorok és destruktorok.

3.10.1 Dinamikus objektumok létrehozása

Dinamikus objektumok létrehozására a C++ `new` operátora szolgál. Tekintsük erre a következő példát:

```
circle *egy_kor = new circle(100,200,50);
```

Ennek hatására az `egy_kor` azonosítójú `circle` típusra mutató pointer megkapja egy `sizeof(circle)` darab byte méretű, `circle` típusú objektum számára lefoglalt tárterület címét. A tárterület-foglalás során végrehajtásra kerül a `circle( )` konstruktor-hívás is. A létrehozott új "kör" középpontjának koordinátái rendre 100 és 200, míg a sugár értéke 50 lesz.

A `new` operátorral tömbök számára is foglalhatunk helyet. A `new`-nak ez az alternatív szintaxisa a következő:

`new` *típus* [*méret*]

A kifejezés értéke az adott típus számára lefoglalt adott méretű tömbre mutató pointer lesz.

A `new`-val történő memória-foglaláskor fellépő hibákat a `_new_handler` pointer (előredefiniált globális változó) által mutatott függvény kezeli le. Alapértelmezésben sikertelen tárterület-foglalási kisérlet esetén `new` visszatérési értéke `NULL`, és programunk futása úgy folytatódhat, mintha mi sem történt volna, ugyanis a `_new_handler`-en keresztül aktivizált hibakezelő nem csinál semmit. Ugyanakkor lehetőségünk van arra, hogy a `_new_handler`-t egy saját hibakezelőre állítsuk a `set_new_handler( )` függvényhívás segítségével. Ezt a következő példa szemlélteti:

```
#include <iostream.h>
#include <stdlib.h>
// ***************************************************
void out_of_store(void)       // Sajat hibakezelo-fv. new-hoz
// ***************************************************
{
        cerr << "operator new failed: out of store\n";
        exit(1);              // cerr-t lasd 3.11.3. alatt
}
// ***************************************************
typedef void (*PF)();         // Fuggvenyre mutato tipus
extern PF set_new_handler(PF);// _new_handler-t allitja
struct pp { double x,y,z,w;   // Nagy struktura. Ennek pro-
            long   a,b,c,d; };// balunk sok helyet foglalni
// ***************************************************
main()          // _new_handler atallitasat szemlelteto program
// ***************************************************
{
  set_new_handler(&out_of_store);
  pp *p = new pp[65535];
  cout << "Meghivtuk new-t, p = " << long(p) << '\n';
}                             // cout-ot is lasd 3.11.3 allatt
// ***************************************************
```

A fenti program soha sem fog "normálisan" lefutni, hanem mindig az *operator new failed: out of store* üzenetet kapjuk a képernyőn és visszatérünk 1-es hibakóddal az operációs rendszerhez.

3.10.2 Dinamikus objektumok megszüntetése

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, hogy a `delete` csak a `new`-val létrehozott dinamikus objektumok megszüntetésére használható. A `delete` oparátornak is van a `new`-hoz hasonló alternatív szintaxisa:

`delete` *objektum* [*méret*]

Megjegyzendő, hogy a C++ nem definiál memória-karbantartó (garbage collector) rendszert, így az "elvesztett" dinamikus objektumokat nem tudjuk megszüntetni.

3.11 További flexibilitás a C++-ban

Az eddigiekben a C++ leglényegesebb vonásait mutattuk be. Ebben a fejezetben további, a nyelvet igen rugalmassá tevő tulajdonságokat ismertetjük.

3.11.1 Rokonok és barátok

Az öröklés következtében különböző őstípusokból kiindulva teljes származási fákat, másképp kifejezve osztályhierarchiákat hozhatunk létre. A többszörös öröklés azt is lehetővé teszi, hogy egy újabb típust több őstípus leszármazottjaként hozzunk létre. Íly módon egymással ronkonságban álló típusok deklarálhatók. Elképzelhető azonban, hogy egy programon belül egymástól teljesen független családfák jönnek létre. Az egy típuscsaládba (származási fába, családfába, osztályhierarchiába) tartozó típusok egymás *rokonai*, míg a különböző családba tartozók egymás számára teljesen idegenek.

Az életben az ember természetesen nem csak 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 kézbentartottuk a mezőhozzáférést az egyes típuscsaládok deklarációjakor, akkor egy adott típus egy függvénymezője nem férhet hozzá egy nem rokon típus adatmezőihez. Márpedig nem zárható ki az az eset, amikor egy ilyen hozzáférés haszonnal járhat. Hogy egy C++ programban ezt a hasznot élvezhessük, a `friend` kulcsszóval megadhatjuk, hogy a programunk nem rokon tárolási egységei közül melyek állnak egymással *barátságban*.

Kicsit egzaktabban: Szükségünk lehet arra, hogy egy adott osztály privát adatmezőit egy olyan függvénnyel is manipulálhassuk, amelyik nem eleme az adott osztálynak. Ezt úgy érhetjük el, hogy az osztály deklarációjakor a `friend` kulcsszót követően felsoroljuk azokat a külső függvényeket, amelyek számára jogot szeretnénk biztosítani az adott osztály adatmezőinek elérésére. A `friend` deklaráció az osztály-deklarációnak akár a `private`, akár a `public` részében is lehet. Lássunk néhány példát a "függvény-barátságra":

```
int ext_func(void) { ... }
```

```
...
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);
           };
```

Az előbbi példában `ext_func` egy normál C függvény. Az `xx` osztály deklarációjakor jogot biztosítottunk számára a `data_1` és `data_2` adatmezők elérésére. Természetesen egy másik osztály függvénymezői is lehetnek barátok:

```
class yy {
            ....
            void yy_func1(void);
            ....
          };
class zz {
            ....
            void zz_func(void);
            friend void yy::yy_func1(void);
            ....
          };
```

Itt tehát az `yy` osztály `yy_func1` nevű függvénymezőjét deklaráltuk a `zz` osztály barátjának. Arra is lehetőségünk van, hogy az `yy` osztályból ne csak egy, hanem mindegyik függvénymezőt `zz` barátjává tegyük:

```
class zz {
            ....
            void zz_func(void);
            friend class yy;
            ....
          };
```

3.11.2 Operator overloading

A C++-nak van egy speciális tulajdonsága, amellyel kevés más nyelv rendelkezik. Nevezetesen az, hogy a nyelv által definiált, létező operátorokhoz

újabb jelentést rendelhetünk. Ez hasznos lehet abból a célból, hogy valamilyen osztályként definiált új adatstruktúrán is elvégeztethessük azt a műveletet, amit az illető operátortól megszoktunk az elemi típusok esetében. 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 változó, 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 rendelhetünk olyan értelemben, hogy azt a bizonyos műveletet több, különböző típusú adaton is végre szeretnénk hajtani – ezt neveztük *overloading*-nak. Gondoljunk bele, hogy az *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 operátora (amelyik mind különböző sorszámozott típusú adatokon, mind pedig lebegőpontos számokon értelmezett művelet), vagy gondoljunk az egyes aritmetikai operátorokra, amelyek a hagyományos C különböző elemi típusain nagyjából egyformán működnek.

Az operátor-overloading lehetőségét a C++ kiterjeszti úgy, hogy az egyes operátorokhoz a felhasználó is megadhat újabb értelmezést egy ún. operátor függvény segítségével. Ez úgy lehetséges, hogy az `operator` kulcsszót követően írjuk azt az operátor-szimbólumot, amelyikhez az újabb jelentést szeretnénk rendelni. Így például az `operator+` annak a függvénynek lesz az azonosítója, amelyikkel a + operátorhoz rendelünk egy újabb jelentést. Nézzünk erre egy példát!

3.11.3 Példa egy operátor új jelentésének definiálására

Definiáljunk egy `set` nevű típust egész számok halmazának ábrázolására és értelmezzük rá a + operátort az unió képzésre! Ehhez tekintsük át a következő oldalakon található mintaprogramot!

```
#include <stdio.h>
// ***************************************************
class set {                  // A halmaz tipus definicioja
// ***************************************************
            int *elems;      // A halmaz elemeire mutat
            int no;          // A halmaz elemeinek szama
    public:
            set(int n=0, int *e=NULL);
            ~set( ) { delete elems; };
```

```
            int get_no( ) { return no; };
            int get_elem(int which)
                        { return elems[which]; };
            set operator+(set arg); // Halmazok unioja
          };
// *********************************************************
set::set(int n,int *e)
// *********************************************************
{
   int i;
   if (n)  // Csak akkor fut, ha nem null-halmaz lesz
   {
     elems = new int[n];
     no = n;
     for (i = 0; i < n; i++) elems[i] = e[i];
   }
}
// *********************************************************
set set::operator+(set arg) // Az unio muvelet
// *********************************************************
{
  int arg_no,*new_set,i,j,k,e,flag;
                    // Egyik operandus elemszama a 'no'
  arg_no  = arg.no; // Masik operandus elemszama
  new_set = new int[arg_no + no];
                    // Az eredmenyt tartalmazo tomb
  k = 0;
  for (i = 0; i < no; i++)
  {
     new_set[k++] = elems[i];
              // Az egyik operandusbol minden elem kell
  }
  for (j = 0; j < arg_no; j++)
  {
     e = arg.elems[j],  flag = 0;
     for (i = 0; i < no; i++)
     {         // A masikbol csak az uj elemek kellenek
        if (flag = (e == elems[i])) break;
     }
     if (!flag) new_set[k++] = e;
  }
  set result(k,new_set);
```

```
                // Eredmeny halmazt konstrualunk
  delete new_set;
                // Az eredmenyt tartalmazo tomb nem kell
  return result;
}
// Adatok az unio muvelet kiprobalasahoz:
static int h1[ ] = { 1, 2, 3, 4 },
           h2[ ] = { 5, 2, 7, 1 };
// ********************************************************
main()
// ********************************************************
{
  int i, no;
  set a(0,NULL),          // 'a' eloszor ures halmaz.
                          // Majd itt lesz 'b+c'
     b(4,h1), c(4,h2); // Ket operandus az uniohoz

  // A ket egyesitendo halmaz elemeit kiiratjuk:

  cout << "Elements of b:\n";
  for(i = 0, no = b.get_no( ); i < no; i++)
   cout << i << ". element = " << b.get_elem(i) << '\n';
  cout << '\n';
  cout << "Elements of c:\n";
  for(i = 0, no = c.get_no( ); i < no; i++)
   cout << i << ". element = " << c.get_elem(i) << '\n';
  cout << '\n';

  // Az 'a' halmazban lesz 'b' unio 'c' :

  a = b + c;

  // Kiiratjuk az eredmenyt:

  cout << "Elements of a:\n";
  for(i = 0, no = a.get_no( ); i < no; i++)
   cout << i << ". element = " << a.get_elem(i) << '\n';
  cout << '\n';
}
// ********************************************************
```

Most tekintsük át az operátor-függvény működését. Az első, amit meg kell említenünk, hogy a `b + c` kifejezés a következőt jelenti:

```
b.operator+(c)
```

Értelemszerűen `operator+` a `b` objektum egyik függvénymezője, tehát a függvénydefiníción belül a `no` és `elems` mezőazonosítók `b` mezőire vonatkoznak, a `this` pointer `b`-re mutat. Az unió művelet második operandusa, `c`, az operátorfüggvényben függvényargumentumként jelentkezik, `arg` néven. Az unió halmaz számára a `new` operátorral létrehozunk egy egész típusú tömböt, `new_set`-et, amelynek mérete a két operandushalmaz méretének az összege. Ez az eredményhalmaz szempontjából elégséges felülbecslés. Az eredményhalmazba először a `this->elems` tömb elemeit egytől-egyig átmásoljuk, majd egy kettős ciklusban `arg.elems`-ből azokat az elemeket, amelyeket a `this->elems` nem tartalmazza. A másoló kettős ciklus végén a `k` számláló tartalmazza az unió-halmaz elemeinek a számát, `new_set` pedig a halmaz elemeit. Ezután létrehozunk egy `result` nevű halmazt `k` elemszámmal és `new_set` elemekkel. Ez a `set` típus konstruktorával történik. Miután megszületett az eredményhalmaz, `new_set`-et a `delete` operátorral megszüntetjük, végül pedig a `return` utasítással `result`-ot visszatérési értékként átadjuk. Hogy szűnik meg a `result`? Nos, emlékezzünk vissza a destruktorokkal foglakozó részre, ahol is azt mondtuk, hogy egy adott objektumra a destruktor automatikusan meghívódik, mihelyt kilépünk egy objektum érvényességi tartományából. Fölmerül az a kérdés is, hogy miért kellett 3 sor erejéig létrehoznunk a `result` változót, miért nem lehetett `this->no`-ba és `this->elems`-be tenni a végeredményt és `*this`-t szolgáltatni visszatérési értékként? A magyarázat egyszerű: mert akkor elrontottuk volna az első operandust, `b`-t. Márpedig a `+` művelettől a halmazok esetében is elvárjuk, hogy a számokon értelmezetthez hasonló módon működjön; ne okozzon kellemetlen mellékhatásokat, mindkét operandusa érintetlen maradjon (és ugyanakkor tudja a halmazalgebrát is).

3.12 C++ I/O könyvtárak

Az `stdio.h` include file-ban deklarált szabványos folyam jellegű formátumozott I/O függvények (mint például `printf`, `scanf`, stb.) mellett a C++ újabb hasonló, folyam-orientált függvényeket tartalmaz az `iostreams` könyvtárban. Az új C++ I/O függvények deklarációját az `iostream.h` include file-ban találhatjuk. Az új I/O könyvtár használata előnyöket nyújt az `stdio.h`-hoz képest. Ezek egyike, hogy a függvények szintaxisa sokkal egyszerűbb, elegánsabb, olvashatóbb. Egy másik tényező, hogy a C++ folyamkezelő mechanizmusa a leggyakrabban előforduló esetekben

hatékonyabb és rugalmasabb, mint a hagyományos C folyamkezelés. A formátumozott kimenet előállítása az operátor-overloadingnak köszönhetően például sokszor egyszerűbb, mint a `printf`-fel. Ugyanazon operátor használható mind előredefiniált, mind pedig a felhasználó által definiált adattípusok kiiratására.

A C++-ban négyelőre definiált folyam jellegű objektum van. Ezek a következők:

- `cin` a szabványos bemenet, amely általában a billentyűzet. A hagyományos C-beli `stdin`-nek felel meg.
- `cout` a szabványos kimenet, amely általában a képernyő. A hagyományos C-beli `stdout`-nak felel meg.
- `cerr` a szabványos hibakimenet, amely általában a képernyő. A hagyományos C-beli stderr-nek felel meg.
- `clog` a `cerr` teljesen bufferelt változata, nincs megfelelője a C-ben.

Ezek a folyamok tetszőleges eszközre vagy file-ba átirányíthatók, míg C-ben csak az `stdin` és az `stdout` irányítható át. Az `iostream` könyvtár elemei hierarchikusan épülnek egymásra. A könyvtár elemei az alap-primitívtől a legspecializáltabb elemig a következők:

- `streambuf` a memóriabuffereket és az azokat kezelő eljárásokat tartalmazza,
- `ios` a folyamok állapotváltozóit és a hibákat kezeli,
- `istream` egy `streambuf`-ból származó formátumozott és kötetlen formátumú karakterfolyam konverzióját végzi,
- `ostream` egy `streambuf`-ba küldendő formátumozott vagy kötetlen formátumú karakterfolyam konverzióját végzi,
- `iostream` `istream`-et és `ostream`-et kombinálja annak érdekében, hogy kétirányú folyamokat lehessen kezelni,
- `istream_withassign` a `cin` folyam számára definiál konstruktorokat és értékadó operátorokat,
- `ostream_withassign` a `cout`, `cerr` és a `clog` folyamok számára definiál konstruktorokat és értékadó operátorokat.

Az **istream** osztály az overloading segítségével egy új jelentést ad a >> operátor számára a standard elemi típusok esetére. Ha az **x** változó típusa standard elemi típus, akkor a **cin>>x** kifejezés mellékhatásaként az **x** típusának (**char**, **int**, **long**, **float**, **double**) megfelelő konverziós input rutin az **x** változónak megfelelő memóriacímre teszi a standard bemenetről érkező adatot. A kifejezés értékeként **cin**-t kapjuk vissza.

Az előbbiekhez hasonló módon, az **ostream** osztályban található a << operátor új értelmezése. A **cout<<x;** utasítás hatására az **x** változó típusának megfelelő output rutin a szabványos kimeneti állományra küldi az **x** értékének megfelelő karaktersorozatot és visszaadja **cout**-ot. (Itt **x** típusa szintén valamelyik standard elemi típus lehet.)

A << és >> operátor-függvények a megfelelő folyam osztályára vonatkozó referencia típusú visszatérési értéket adnak, így több ilyen operátor láncba fűzhető:

```
int    i = 0, x = 243;
double d = 0;
cout << "The value of x is " << x << '\n';
cin >> i >> d;  // Egy int-et, utana SPACE-t,
                // majd egy double-t var.
```

3.13 OOP megközelítésű rendszerfüggvények

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 részletesebben.

3.13.1 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** include file-ban:

```
struct complex
{
```

```
        double x, y;
    }
```

amely 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 műveletet nem igen végezhetünk, mert hiányoznak a megfelelő definíciók. A problémát a C++ által nyújtott operátor-overloading oldja meg. Ha a `complex.h` file-t 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 >>-t és a <<-t)
- a gyakran használt matematikai függvényeket, például a gyökvonást (`sqrt`), a logaritmust (`log`), stb.

Komplex argumentum esetén a kérdéses függvénynek a komplex argumentumú változata aktivizálódik. Például ha a -1-nek a komplex gyökét keressük, akkor az

```
sqrt(complex(-1))
```

írásmód vezet helyes eredményre, míg `sqrt(-1)` esetén programfutás közben a *Domain error* hibaüzenetet kapjuk.

3.13.2 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 a 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 mégis 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 a `bcd.h` fejléc file-ban deklarált `bcd` típus által. Így például a 0.01 is pontosan ábrázolható, ha `bcd` típusú változóban 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 az `bcd`-ben is csak kerekítve ábrázolható.
- A gyakran használt matematikai függvényeknél – ilyen például a `sqrt` és a `log` – a `bcd` argumentum felülíródik.
- 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` változta kerül végrehajtásra. A `bcd` típusnak (amelyik természetesen egy `class`), több konstruktora létezik. Ezek segítségével állíthatunk elő `bcd` számokat különböző elemi típusú kifejezésekből.

A bcd-bináris konverzió számára megadható a figyelembe veendő decimális jegyek száma. Például:

```
1000.00/7                = 142.85714...
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
```

KIADÓI SZOLGÁLTATÓ ÉS KERESKEDŐ KFT

COMPUTERBOOKS

1126 BUDAPEST, TARTSAY VILMOS U. 12.
TEL.: 1751 564, 1753 591
FAX.: 1757 929

E kötetnek folytatását tervezzük

WINDOWS programok fejlesztése

BORLAND C++ környezetben

címmel, melyet igényes programozóknak szánunk.

A könyvhöz lemezen példaprogramokat

mellékelünk.

Megjelenik 1991. decemberében.

4 Fejezet

IBM PC specifikus lehetőségek

A BORLAND C++ rendszer – a szábványos ANSI C, illetve C++ könyvtárak mellett – nagyon gazdag rutinkészlettel rendelkezik, amely biztosítja, hogy teljes mértékben kihasználhassuk az IBM PC nyújtotta lehetőségeket.

Egyrészt DOS, illetve BIOS rutinhívásokat használhatunk, ugyanakkor a BORLAND C++-ban nagyon hatékonyan megvalósított, magasabb szintű függvények segítségével kezelhetjük az IBM PC képernyőjét akár szöveges, akár grafikus üzemmódban. A BORLAND C++ azt is lehetővé teszi, hogy – bár csak ún. overlay szervezéssel – kihasználhassuk az IBM PC 640 kbyte-on felüli memóriáját, legyen az akár ún. extended, vagy expanded memória.

Természetesen az itt ismertetésre kerülő lehetőségeket viszonylag egyszerű, gyorsan megírandó programok elkészítéséhez ajánljuk. Komolyabb igényű felhasználói felületek, vagy nagy tárigényű programok készítésekor az MS-Windows használatát javasoljuk. Ezt természetesen szintén támogatja a BORLAND C++, de az MS-Windows alkalmazói programok írásával egy külön kötet foglalkozik.

4.1 Szöveg és grafika

A BORLAND C++ grafikus függvényeinek igen gazdag könyvtára lehetővé teszi, hogy különféle ábrákat, diagramokat rajzoljunk a képernyőre. Az IBM PC képernyőjét *text* (szöveges) üzemmódban, vagy *grafikus* üzemmódban használhatjuk. A képernyőt e kétfajta üzemmódban azonban

csak felváltva lehet működtetni.

Egy IBM PC-nek többfajta video adaptere lehet. A *Monochrome Display Adapter* (MDA) csak szöveges üzemmódban használható, míg a *Color Graphics Adapter* (CGA), a *Enhanced Graphics Adapter* (EGA), a *Video Graphics Array* (VGA) vagy a *Hercules Monochrome Graphics Adapter* nevű kártyák akár szöveges, akár grafikus módban használhatók. Minden adapternél 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.(Lásd az A függeléket.)

- Grafikus üzemmódban a PC képernyője képpontokra (*pixel*) van osztva. A képpontok száma és szinezési 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 koordinátja (1,1), az x koordináták balról jobbra, az y koordináták pedig 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 alkamazói program a képernyőre ír például a `cprintf` függvénnyel, az eredmény 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 a szintén automatikus

vágás történik. Amikor tehát rajzolunk a grafikus ablakba, és egyes vonalak az ablak határain 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 `setviewport` függvénnyel állíthatjuk be.

4.1.1 Programozás szöveges üzemmódban

Röviden összefoglaljuk azokat 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 szöveg-attribútum (szín, háttérszín, fényerősség, stb) kezelő függvények. Ezek prototípusai a `conio.h` include file-ban 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 4 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` stringet küld a képernyőre
- `getche` olvas egy karaktert és kiírja a képernyőre
- `putch` egyetlen karaktert küld a képernyőre

Szöveg és kurzor mozgatása a képernyőn:

- `clrscr` törli a szöveg képernyő ablakot
- `clreol` törli a sort a kurzor pozíciótól 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

Szövegblokkok másolása:

- **gettext** szöveget másol egy képernyő területről 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 a 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 szövegattribútumok tárolása 8 biten történik. Egy karakterpozicióhoz 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 `text_info` 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` file-ban van deklarálva) az alábbi információkkal:

- az aktuális video mód
- az ablak helyzete abszolút képernyő koordinátákban
- az ablak méretei
- az aktuális előtér és háttér színe
- a kurzor aktuális pozíciója

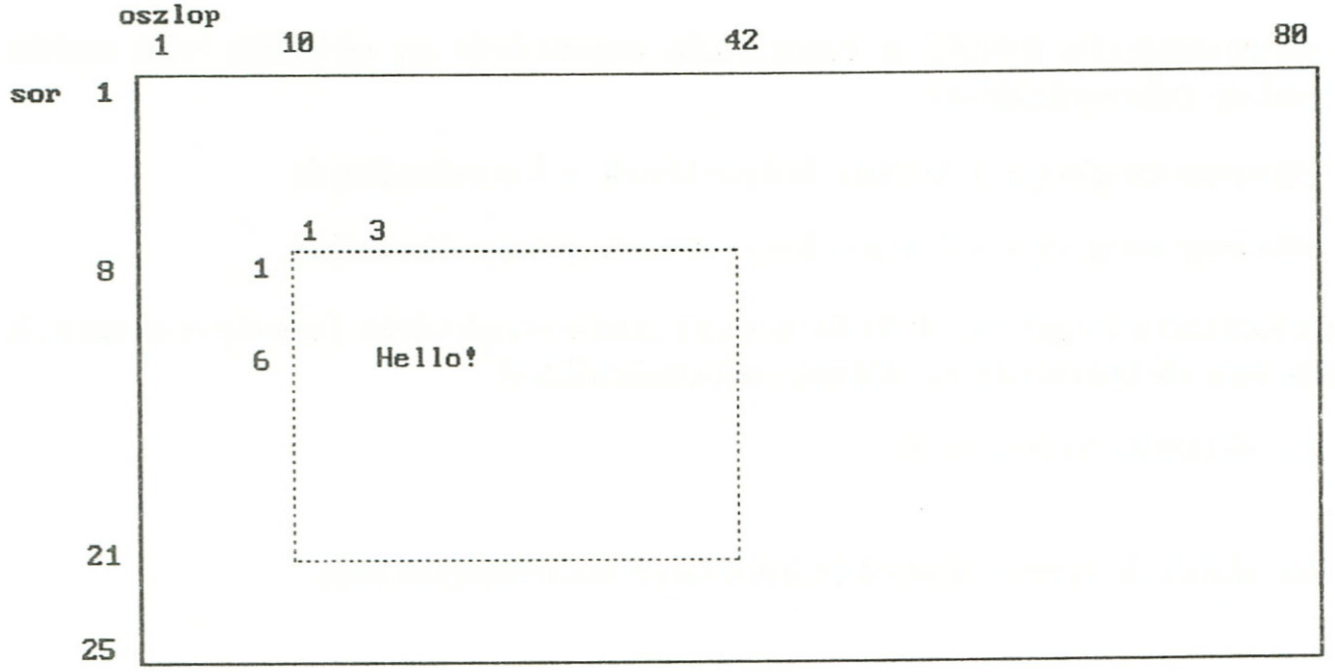
Lekérdezhetjük a kurzor az ablakhoz bal felső sarkához viszonyított pozícióját a `wherex` és a `wherey` függvényekkel. A `_setcursortype` függvény segítségével meg tudjuk változtatni a kurzor alakját. A függvény bemenő paramétere határozza meg a kurzor új alakját. E paraméter értéke háromféle lehet:

`_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 definiáljunk egy $80 \cdot 25$ méretű ablakban egy kisebb ablakot, amelynek az "origója" (bal felső sarka) a (10,8) koordinátájú karakterpozición 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 pozición 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_modes** típusú. Ez egy felsorolt típus, amely a **conio.h** include file-ban van definiálva. A **text_modes** típus lehetővé teszi az üzemmód kiválasztásnál a szimbolikus névvel történő megadást. A **textmode** függvény **mode** paramétere számára definiált szimbolikus értékek a következők:

Szimbolikus 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 szimbolikus nevei és konstansai a következők:

Szín	Szimbólum	Érték	Előtér vagy háttér
fekete	BLACK	0	mindkettő
kék	BLUE	1	mindkettő
zöld	GREEN	2	mindkettő
türkiz	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 hozzá 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 bitorientált függvényekig (mint például a **getimage** és a **putimage**).

Az eljárások között van pont, vonal, kör, körív, ellipszis, ellipszisív, téglalap, poligon és téglatest rajzoló, de 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 kiirandó 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 könyvtárban vannak, a deklarációk és konstansok a `graphics.h` file-ban találhatók. A grafikus csomag használatához szükség van a megfelelő grafikus meghajtókra (`.bgi file-ok`) és a felhasznált karakter készleteket tartalmazó `.chr` file-okra. A grafikus függvényeket az alábbi módon használhatjuk:

- Ha az integrált környezetben (*IDE*) dolgozunk, akkor a *Full Menus* opció legyen *On*-ba kapcsolva. Ellenőrizzük, hogy az **O**ption|**L**inker|**G**raphics *library* opciót kiválasztottuk-e. Amennyiben ez az opció be van kapcsolva, a BORLAND C++ automatikusan hozzászerkeszti a grafikus könyvtárat a programunkhoz.

- Ha a `BCC` vagy `BCCX` parancssor orientált fordítót használjuk, akkor a parancssorba fel kell tüntetni a `graphics.lib` könyvtár nevét. Ha például programunk neve `sajat.c`, és BGI grafikát használunk benne, akkor a `BCC`-t a következőképpen hívjuk meg a DOS-ból:

```
BCC SAJAT GRAPHICS.LIB
```

Mivel a grafikus függvények `far` pointereket használnak, ezért grafika esetén nem használhatjuk a `tiny` tármodellt.

A BORLAND C++ grafikus rendszere úgy működik, hogy az adott grafikus kártya kezeléséhez szükséges programrészeket a megfelelő `.bgi` file-ból futási időben tölti be a tárba. A programunk által használt kerekterfontokat is hasonlóképpen, futási időben olvassa be a futtató rendszer a megfelelő `.chr` file-ból.

Lehetőségünk van azonban arra, hogy akár egy `.bgi` file-t, akár egy `.chr` file-t `.obj` file-lá alakítsunk, és elhelyezzük a `graphics.lib` könyvtárban. Ekkor – bizonyos feltételek mellett – a grafikus meghajtókat és a programunk által használt karakterkészleteket a programunkba szerkeszti a BORLAND C++.

Az `.obj` file-ba konvertálást a `BGIOBJ` segédprogrammal végezhetjük el. A keletkezett `.obj` file-okat a `TLIB` programmal adhatjuk hozzá a `graphics.lib` állományhoz. E programok használatáról a `UTIL.DOC` file ad bővebb tájékoztatást.

Az IBM PC többfajta grafikus kártyával rendelkezhet, amelyek közül BORLAND C++ az alábbi változatokat támogatja:

Grafikus meghajtó szoftver	Grafikus kártya
CGA.BGI	IBM CGA, MCGA
EGAVGA.BGI	IBM EGA, VGA
HERC.BGI	Hercules egyszínű
PC3270.BGI	IBM 3270
IBM8514.BGI	IBM 8514
ATT.BGI	IBM AT&T 400

Egy grafikát használó alkalmazói program fordításához és futtatásához a forrásprogram mellett tehát az alábbi file-ok is szükségesek:

- `graphics.lib` könyvtár
- a megfelelő grafikus meghajtó (`.bgi` file)
- karakterkészlet használata esetén a megfelelő font-file (`.chr` file)

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 lekérdezés,

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
- **getmoderange** visszatér a specifikált meghajtó legkisebb és és legnagyobb felbontásával
- **initgraph** inicializálja a grafikus rendszert, a hardvert grafikus módba helyezi
- **installuserdriver** installálja az felhasználó által írt grafikus meghajtókat
- **installuserfont** betölti a felhasználó által készített karakter készletet a BGI karakter file táblába
- **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** file-t) és a rendszert a megfelelő grafikus módba helyezi.

Rajzolás:

- **arc** körivet 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 vizszintes/függőleges képarányát adja meg
- **getlinesettings** az aktuális vonal stílusát, mintáját és vastagságát adja meg
- **line** egyens szakaszt rajzol (x0,y0)-ból (x1,y1)-be.
- **linerel** egyenes szakaszt rajzol egy pontba, relatív távolsággal a kurzor pozíciójától

- `lineto` egyenes szakaszt rajzol az aktuális poziciótól 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 képarrányt
- `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
- `floofill` 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 elliptikus 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 képernyő tartományt memóriba menti ki
- `imagesize` megadja a kijelölt téglalap alakú tartomány byte-jainak 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

Szövegkiirás 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 karakterkészletet, 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:

- `getbkcolor` megadja az aktuális háttérszínt
- `getcolor` megadja az aktuális rajzolási színt
- `getdefaultpalette` kitölti a paletta struktúrát
- `getmaxcolor` az aktuális grafikus módban használható színek maximálisan értékét adja meg
- `getpalette` az aktuális palettát és annak méretét adja meg

- `getpalettesize` 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 argumentumban megadottak 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 `graphresult` függvénnyel lekérdezhetjük. Ha a hibakód nulla, akkor sikeres volt a grafikus függvény végrehajtása. A `graphresult` függvénynek az alábbi kód konstansai léteznek:

Szimbólum	Magyarázat
`grOk`	Nincs hiba
`grNoinitGrapgh`	Nincs `.bgi` file installálva.
`grNotDetected`	Nincs grafikus kártya
`grFileNotFound`	A `.bgi` file hiányzik
`grInvalidDriver`	Érvénytelen grafikus meghajtó
`grNoLoadMem`	Kevés a memória driver számára
`grNoScanMem`	Kevés a memória a vizsgálathoz
`grNoFloodMem`	Kevés a memória a kitöltéshez
`grFontNotFound`	Nem létező karakterkészlet file
`grNoFontMem`	Kevés a memória a font számára
`grInvalidMode`	A kiválasztott grafikus mód érvénytelen
`grError`	Grafikus hiba
`grIOError`	Grafikus I/O hiba
`grInvalidFont`	Érvénytelen karakterkészlet file
`grInvalidFontNum`	Érvénytelen karakterkészlet azonosító
`grInvalidDeviceNum`	Érvénytelen egység szám
`grInvalidVersion`	Érvénytelen file verzió

- `getpalettesize` 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 argumentumban megadottak 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 `graphresult` függvénnyel lekérdezhetjük. Ha a hibakód nulla, akkor sikeres volt a grafikus függvény végrehajtása. A `graphresult` függvénynek az alábbi kód konstansai léteznek:

Szimbólum	Magyarázat
`grOk`	Nincs hiba
`grNoinitGrapgh`	Nincs `.bgi` file installálva.
`grNotDetected`	Nincs grafikus kártya
`grFileNotFound`	A `.bgi` file hiányzik
`grInvalidDriver`	Érvénytelen grafikus meghajtó
`grNoLoadMem`	Kevés a memória driver számára
`grNoScanMem`	Kevés a memória a vizsgálathoz
`grNoFloodMem`	Kevés a memória a kitöltéshez
`grFontNotFound`	Nem létező karakterkészlet file
`grNoFontMem`	Kevés a memória a font számára
`grInvalidMode`	A kiválasztott grafikus mód érvénytelen
`grError`	Grafikus hiba
`grIOError`	Grafikus I/O hiba
`grInvalidFont`	Érvénytelen karakterkészlet file
`grInvalidFontNum`	Érvénytelen karakterkészlet azonosító
`grInvalidDeviceNum`	Érvénytelen egység szám
`grInvalidVersion`	Érvénytelen file verzió

Állapotlekérdezés:

- `getarcoords` az utoljára rajzolt körív vagy ellipszis ív koordinátáinak értékét adja meg
- `getaspectratio` a grafikus képernyő oldalarány értékét adja meg
- `getbkcolor` az aktuális háttér színét adja meg
- `getcolor` az aktuális rajz színét adja meg
- `getdrivername` az aktuális grafikus meghajtó nevét adja vissza
- `getfillpattern` 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 meg
- `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 meg
- `getmoderange` az adott meghajtó módhatárait adja meg
- `getpalette` a palettát és méretét adja meg
- `getpixel` az (x,y) képpont színét adja meg
- `gettextsettings` az aktuális karakterkészletet, irányát, méretét és helyzetét adja meg
- `setviewsettings` információt ad az aktuális rajzolási ablakról
- `getx` az aktuális kurzorpozíció x koordináta értékét adja meg
- `gety` az aktuális kurzorpozíció y koordináta értékét adja meg

Színkezelés különböző meghajtók esetén:

- EGA esetén a hardver 64 színt tud kezelni, azonban csak 16 színt használhatunk egyidejüleg.
- 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 3 színt a rajzoláshoz. A mód kiválasztásával aktiváljuk a palettát, amikor a `CGAC0`, `CGAC1`, `CGAC2` vagy a `CGAC3` között választunk.

Paletta szám	1	2	3
0	`CGA_LIGHTGREEN`	`CGA_LIGHTRED`	`CGA_YELLOW`
1	`CGA_LIGHTCYAN`	`CGA_LIGHTMAGENTA`	`CGA_WHITE`
2	`CGA_GREEN`	`CGA_RED`	`CGA_BROWN`
3	`CGA_CYAN`	`CGA_MAGENTA`	`CGA_LIGHTGRAY`

Például, ha a 3-as palettát választottuk, azaz a grafikus üzemmódnak a `CGAC3`-at adtuk meg, akkor az alábbi színeket használhatjuk:

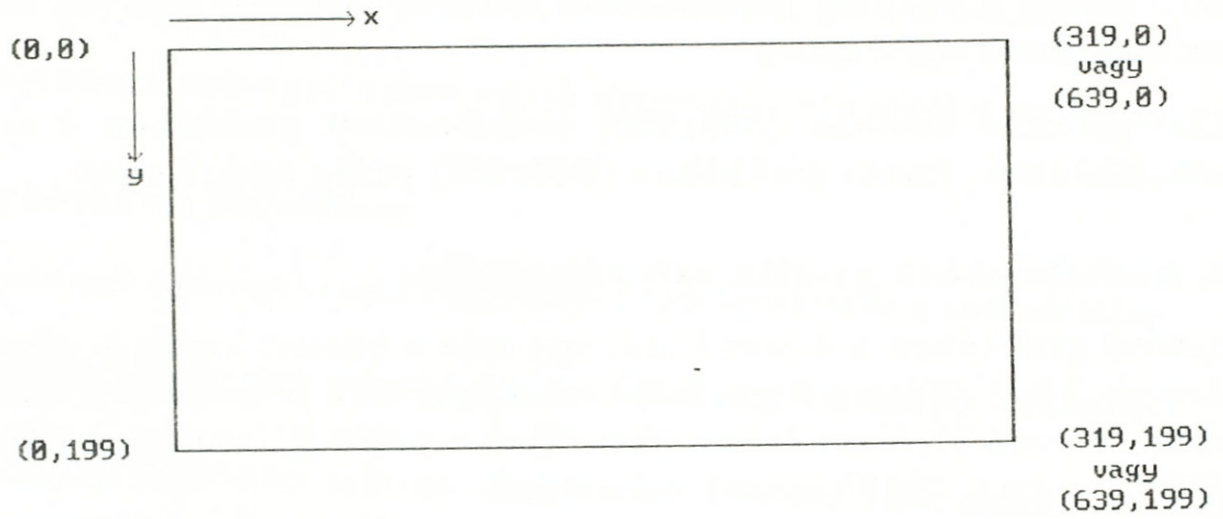
```
CGA_CYAN      CGA_MAGENTA      CGA_LIGHTGRAY
```

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 kéké 1, stb. A **graphics.h**-ban definiált konstansokat felhasználva például a `EGA_BLACK` feketét, a `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 lekérdezhet ők a **getmaxx** és a **getmaxy** függvényekkel. Például CGA meghajtó esetén $320 \cdot 200$ pont van színes



4.2. ábra: Koordináták a grafikus képernyőn

(4 szín használata esetén) üzemmódban, fekete-fehér üzemmóban pedig $640 \cdot 200$ (lásd a 4.2. ábrát).

4.2 Overlay a BORLAND C++-ban

Az ún. *overlay* technika segítségével oldhatjuk meg azt, hogy a rendelkezésünkre álló operatív tár méreténél nagyobb programokat is futtathassunk. A dolog lényege, hogy a program egy része állandóan (rezidensen) a memóriában van, más részei pedig a fennmaradó memóriaterületen osztoznak úgy, hogy időben váltják egymást. Tehát amíg egy nem rezidens programszegmens – ún. *overlay szegmens* az operatív tárban van, addig a többi hasonló szegmens egy háttértároló eszközön várakozik. Ebből következik, hogy a szegmensváltások miatt egy overlay szervezésű program futási ideje jelentősen megnőhet.

A BORLAND C++ overlay kezelő rendszere, az ún. **VROOMM** (*Virtual Runtime Object-Oriented Memory Manager*) végzi magas szinten a szervezést. Egy hagyományos overlay rendszerben a szubrutinok overlay egységeket ún. unitokat alkotnak. Egy uniton belül az egyes szubrutinok hívhatják egymást, de nem hívhatnak más unitbeli függvényeket. Az overlay unitok felülírják egymást, egyszerre csak egy tartózkodik a memóriában és mindegyikük ugyanazt a memóriaterületet foglalja el.

A **VROOMM** rendszer dinamikus szegmens kezelést tesz lehetővé. A alap csereegység a szegmens. A szegmensek egy vagy több object-modulból állhatnak. Nagyon fontos, hogy bármelyik szegmensbeli függvény hívhat bármely más szegmensbeli függvényt. A **VROOMM** a memóriát két részre osztja, egy alapmemóriára és egy ún. *swap* területre. Bármikor, ha egy szegmensből egy olyan függvényt hívunk, amely nincs sem azalapmemóriában, sem a *swap* területen, akkor a *swap* területre betöltődik a hívott függvényt tartalmazó másik szegmens. Ha a szegmensből nem aktivizálnak egy függvényt sem, akkor a szegmens inaktív, tehát kikerülhet a memóriából és helyette egy másik szegmens tölthető be. Ez a *dinamikus cserélő (swapping) módszer.* Természetesen minél nagyobb az ún. *swap* memória, annál gyorsabban fut az overlay szervezésű program. A **VROOMM** rendszer az overlay buffer mérete számára felvesz egy becsült értéket, és ezt a `_ovrbuffer` globális változóban tárolja. Ezt az értékét meg is változtathatjuk. Ha nem elég a **VROOMM** számára hozzáférhető memória, akkor vagy a *Program too big to fit in memory* hibajelzést kapjuk a DOS-tól, vagy a *Not enough memory to run program.* üzenetet kapjuk a program indító kódjától.

A BORLAND C++ overlay kezelő rendszere a következőket nyújtja:

- Minimalizálja a rezidens kódot,
- Beállítja az optimális szegmens méreteket (128 Kbyte területtel kezd és úgy változtat fel/le hogy a sebesség/méret arány gazdaságos legyen).

Az overlay szerkesztéséhez néhány egyszerűbb szabályt figyelembe kell vennii:

- A szegmens egy programnak az a legkisebb része, amely overlay betöltésére alkalmas.
- Az overlay csak a *medium*, *large* és huge modellben működik, nem használható *tiny*, *small* és a *compact* modellben.

4.2.1 Az overlay használata

Ha overlay programot akarunk készíteni a `BCC`-vel, akkor minden modulját `-Y` opcióval kell fordítanunk. Ha részmodult szeretnénk betenni az overlaybe, akkor `-Yo` opcióval kell fordítanunk. Az `-Yo` opció az összes modulra és könyvtárra vonatkozik. Az `-Yo-` opció a `BCC` parancssorában megszünteti az `-Yo` opció hatását. Példaként fordítsuk le overlay szegmensként a `sajat.c` modulunkat, de a `graphics.lib` könyvtárnál tekintsünk el az overlay-től.

Ehhez a

```
BCC -ml -Yo sajat.c -Yo- graphics.lib
```

parancsot kell kiadnunk a DOS-ban. Tekintsünk egy másik példát, ebben az esetben az overlay program álljon 3 modulból. Ezek: `prog.c`, `fg1.c` és `fg2.c`. A `prog.c` feladata következtében kell, hogy állandóan a memóriában maradjon, csak az `fg1.c` és `fg2.c` modulokat tudjuk overlay-ként fordítani:

```
BCC -ml -Y sajat.c -Yo fg1.c fg2.c
```

4.2.2 Swapping

Ha expanded vagy extended memória áll rendelkezésre, akkor az overlay vezérlőnek felajánlható swap memóriaként. Ebben az esetben, ha az overlay vezérlő egy modult eltávolít a memóriából (mivel új modult tölt be és a buffer tele van), akkor az eltávolítandó modult ebbe a memóriába teheti be, így nem kell azt háttértárolóra írnia és később ennek a modulnak az aktivizálására kevesebb idő szükséges, mintha diszkről kellene betölteni.

Mindkét esetben két lehetőség van: az overlay vezérlő automatikusan érzékeli az expanded vagy extended memória jelenlétét, azonban ez nem mindig sikeres, mivel RAM diszk programok minden jelzés nélkül is használhatják az extended memóriát. Ez a probléma elkerülhető, ha megmondjuk az overlay vezérlőnek az extended memória címét és hogy mennyit használhat el belőle.

Expanded memória

Az `_OvrInitEms` függvény kezdőértéket ad az expanded memória használatára. Deklarációja:

```
int far _OvrInitEms(unsigned emsHandle,
                    unsigned emsFirst,
                    unsigned emsPages)
```

Ha az `emsHandle` paraméter értéke zérus, akkor az overlay vezérlő ellenőrzi az extended memória jelenétét és lefoglalja a méretét (ha tudja), amely tartalmazhatja az összes overlay szegmenst, leszámítva az overlay buffer méretét. Máskülönben `emsHandle` tartalmazzon egy legális EMS leírót (*handle*), az `emsFirst` az első használható EMS lapot és az `emsPages`

az overlay vezérlő által használható lapok számát. Ha a függény zérussal tér vissza, akkor az expanded memória hozzáférhető.

Extended memória

Az `_OvrInitExt` függvény kezdőértéket ad az extended memória használatára. Deklarációja:

```
int far _OvrInitExt(unsigned long extStart,
                    unsigned long extLenght)
```

Ha az `extStart` paraméter értéke zérus, akkor az overlay vezérlő teszteli az extended memóriát. Ha lehet, akkor az overlay vezérlő felhasználja a szabad memóriát, amely tartalmazhatja az összes overlay szegmenst, leszámítva az overlay buffer méretét. Máskülönben az `extStart` paraméter tartalmazza a használható extended memória kezdetét, és az `extLength` pedig a byte-ok számát, amely használható az overlay vezérlő számára. Ha az extLength értéke zérus, akkor az overlay vezérlő felhasználja az összes szabad memóriát onnan, ahonnan az `extStart` paraméter megadta. Ha a függvény zérussal tér vissza, akkor az extended memória hozzáférhető. Az `OvrInitExt` a `dos.h` file-ban van deklarálva.

Az extended memória használatára nincs egységes szabvány. Így az overlay vezérlő megpróbál minden ismeretes eljárást arra, hogy megállapítsa az extended memória szabad kapacitását. Ezt a függvényt óvatosan használjuk. Például, ha 2 Mbyte hard diszk cache memória van installálva (ami extended memóriaként használható) az alábbi utasítással az overlay vezérlő a maradék memóriát használhatja:

```
if (_OvrInitExt(1024L * (2048 + 1024), 0L)
   puts("Too little memory!");
```

A Függelék

Include file-ok és függvények

A BORLAND C++ futási idejű könyvtára (*Run-time library*) több, mint 450 rutint (függvényt, makrót) tartalmaz. Ezek egyrésze a rendszerspecifikus tulajdonságokat rejti szabványos kezelői felület mögé (például be/ki és file-műveletek, időkezelés, tárfoglalás, folyamatvezérlés stb.), más részük gyakran használt feladatokat lát el (karakterlánc műveletek, adatkonverziók, diagnosztikai rutinok stb.). A futási idejű rutinok a `Cx.LIB`, `CPx.LIB`, `MATHx.LIB`, valamint az `OLDSTRx.LIB`, `EMU.LIB`, `FP87.LIB`, `IMPORT.LIB`, `OVERLAY.LIB`, `CWIN.LIB` és a `GRAPHICS.LIB` nevű könyvtárakban vannak tárgykódú formában, és a linker onnan emeli be a szükséges modulokat. A BORLAND C++ hat különböző könyvtári modellt kezel. Az `x` helyén a megfelelő memóriamodell kezdőbetűje szerepel, a *tiny* és *small* modellek közös könyvtárat használnak. A függelék első részében áttekintjük a include file-okat, majd a könyvtári rutinok funkció szerinti csoportosítása következik.

A.1 A BORLAND C++ include file-jai

A include file-ok 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 által ajánlott include file elnevezéseket és tartalmakat. Az ANSI által definiált header file-ok (include file-ok) csillaggal (*) vannak az alábbi listában megjelölve.

- `alloc.h` Memóriakezelő függvényeket deklarál (tárfoglalás, memória felszabadítás stb.).
- `assert.h*` Definiálja az `assert` hibakereső makrót.
- `bcd.h` Deklarálja a `bcd` C++ osztályt, a hozzátartozó operátorokat és matematikai függvényeket.
- `bios.h` Az IBM-PC ROM BIOS rutinok hívásában használt különféle függvényeket deklarál.
- `complex.h` Deklarálja a `complex` C++ osztályt, a hozzátartozó operátorokat és matematikai függvényeket.
- `conio.h` A DOS konzol I/O rutinok hívásában használt különféle függvényeket deklarál.
- `ctype.h*` Tartalmazza azokat az információkat, melyeket a karakter osztályozó és karakter konverziós makrók használnak (ilyen az pl. `isalfa` és a `toascii`).
- `custcntl.h` Ez a include file az MS-Windows alkalmazói programok számára tartalmaz definíciókat a Windows felhasználói felületek dialógus dobozaira vonatkozólag.
- `dde.h` MS-Windows alkalmazói programok írásához szükséges definiciókat tartalmaz.
- `dir.h` A katalógusok és elérési útvonal (path) nevek kezeléséhez tartalmaz struktúra definíciókat, makrókat és függvény - deklarációkat.
- `direct.h` A fenti `dir.h` file-t építi be, és definiálja a `_getdrive` makrót.
- `dos.h` Szimbólumdefiníciókat és deklarációkat tartalmaz a DOS és 8086 specifikus rendszer hívások számára.
- `drivinit.h` Az MS-Windows printer driver rutinjainak deklarációját tartalmazza.
- `errno.h*` Mnemonikus konstansokat definiál a standard hibakódok számára.
- `fcntl.h` Szimbolikus konstansokat definiál az `open` könyvtári függvény számára.
- `float.h*` Paramétereket tartalmaz a lebegőpontos rutinok számára.

- `fstream.h` Deklarálja a C++ file I/O-hoz szükséges stream osztályokat.
- `generic.h` Makrókat tartalmaz a `generic` osztály deklarációja számára.
- `graphics.h` A BORLAND C++ saját grafikus könyvtárához definiál konstansokat és deklarálja a grafikus függvények prototípusait.
- `io.h` Struktúrákat és deklarációkat tartalmaz az alacsony szintű input/output rutinok számára.
- `iomanip.h` Deklarálja a C++ I/O manipulátorokat és makrókat tartalmaz paramétezett manipulátorok létrehozásához.
- `iostream.h` Deklarálja a C++ (2.0 verzió) alapvető stream I/O rutinokat.
- `limits.h*` Környezeti információkat, az adott implementációra jellemző korlátokat és a sorszámozott típusok értéktartományának határait adja meg szimbolikus konstansok formájában.
- `locale.h*` Deklarálja azokat a függvényeket, amelyek ország- és a nyelvspecifikus információkat adnak.
- `math.h*` Deklarálja a matematikai függvényeket, definiálja a `HUGE_VAL` makrót és a `matherr` függvény által használt struktúrát.
- `mem.h` Deklarálja a memóriakezelő rutinokat. (Ezek nagyrésze deklarálva van a `string.h`-ban is.)
- `memory.h` Ugyanaz, mint a fenti `mem.h` file.
- `new.h` A C++ `new` operátorának hibakezelőjével kapcsolatos definíciókat tartalmazza.
- `process.h` Struktúrákat és a deklarációkat tartalmaz a `spawn...` és az `exec...` függvénycsaládok számára.
- `setjmp.h*` Definiálja a `jmp_buf` típust, amit a `longjmp` és a `setjmp` függvények használnak, valamint deklarálja ezeket a függvényeket.
- `share.h` Az osztott file-kezelést végző függvények számára definiál paramétereket.
- `search.h` Különböző rendező-kereső algoritmusokat megvalósító függvények (`bsearch`, `lfind`, `lsearch`, `qsort`) prototípusait tartalmazza.

- `signal.h*` Szimbolikus konstansokat és deklarációkat tartalmaz a `signal` és a `raise` függvények használatához.
- `stdarg.h*` Makrókat definiál a változó számú paraméterrel meghívható függvények argumentumlistáinak kezeléséhez.
- `stddef.h*` Különböző közhasznú adattípusokat és makrókat definiál.
- `stdio.h*` A Kerninghan és Ritchie által definiált, és a UNIX System V alatt kiterjesztett standard I/O csomag adattípusait és makróit definiálja. Itt találhatók a szabványos, előredefiniált adatfolyam azonosítók (`stdin`, `stdout`, `stdprn` és `stderr`) és az adatfolyam jellegű I/O műveleteket végző függvények deklaráció.
- `stdiostr.h` Deklarálja a C++ (2.0-ás verzió) `stdio FILE` struktúrák használatához szükséges stream osztályait.
- `stdlib.h*` Általános célú függvényeket deklarál, mint például a konverziós rutinok, a kereső/rendező rutinok stb.
- `stream.h` Deklarálja a C++ (1.2 verzió) I/O stream kezelő rutinjait.
- `string.h*` Deklarálja a különböző sztring- és memóriakezelő rutinokat.
- `strstrea.h` Stream jellegű C++ osztályokat deklarál byte tömbök használatához.
- `sys\stat.h` File-ok megnyitásához illetve létrehozásához szükséges szimbólumokat definiál.
- `sys\timeb.h` Deklarálja az `ftime` függvényt és az általa visszaadott `timeb` struktúrát.
- `sys\types.h` Deklarálja a `time_t` típust, melyet az időkezelő függvények használnak.
- `time.h*` 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.
- `values.h` Fontos konstansokat (köztük gépfüggőeket) definiál a UNIX System V. operációs rendszerrel való kompatibilitás érdekében.
- `varargs.h` Ugyanaz, mint `stdarg.h`. Az ANSI C kompatibilitás biztosítása végett a `stdarg.h` használatát javasoljuk.

- `windows.h` Az MS-Windows SDK rutinjainak definícióit, és a használatukhoz szükséges típus és konstansdefiníciókat tartalmazza.

A.2 A könyvtári rutinok csoportosítása

A BORLAND C++ könyvtári függvények a feladatok egész sorát hajtják végre. Ebben a részben felsoroljuk a rutinokat a deklaráló include file-okkal együtt. A függvények és a makrók feladatok szerinti csoportosításban szerepelnek.

A.2.1 Karakter osztályozó rutinok

Ezek a makrók végzik az ASCII karakterek számjegyekre, vezérlő karakterekre, írásjelekre, kis- és nagybetűkre stb. való osztályozását.

isalnum	ctype.h	islower	ctype.h
isalpha	ctype.h	isprint	ctype.h
isascii	ctype.h	ispunct	ctype.h
iscntrl	ctype.h	isspace	ctype.h
isdigit	ctype.h	isupper	ctype.h
isgraph	ctype.h	isxdigit	ctype.h

A.2.2 Konverziós rutinok

Ezek a rutinok konvertálják a karaktereket és sztringeket különböző numerikus reprezentációkba – lebegőpontos, rövid és hosszú egész típusokba – és vissza, ill. kisbetűből nagybetűbe és vissza.

atof	stdlib.h	strtol	stdlib.h
atoi	stdlib.h	strtoul	stdlib.h
atol	stdlib.h	_tolower	ctype.h
ecvt	stdlib.h	toascii	ctype.h
fcvt	stdlib.h	tolower	ctype.h
gcvt	stdlib.h	_toupper	ctype.h
itoa	stdlib.h	toupper	ctype.h
ltoa	stdlib.h	ultoa	stdlib.h
strtod	stdlib.h		

A.2.3 Katalóguskezelő rutinok

Ezek segítségével kezelhetjük a katalógusokat és az elérési útvonal (path) neveket.

chdir	dir.h	getdisk	dir.h
findfirst	dir.h	mkdir	dir.h
findnext	dir.h	mktemp	dir.h
fnmerge	dir.h	rmdir	dir.h
fnsplit	dir.h	searchpath	dir.h
getcurdir	dir.h	setdisk	dir.h
getcwd	dir.h		

A.2.4 Diagnosztikai rutinok

Ezek a rutinok a beépített hibakezeléshez nyújtanak lehetőségeket.

assert	assert.h	perror	errno.h
matherr	math.h		

A.2.5 Grafikai rutinok

Ezek a függvények a BORLAND C++ saját grafikus könyvtárának részei. A grafikus és szöveges üzemmódú képernyőkezelést teszik lehetővé.

arc	graphics.h	getdefaultpalette	graphics.h
bar	graphics.h	imagesize	graphics.h
bar3d	graphics.h	initgraph	graphics.h
circle	graphics.h	installuserdriver	graphics.h
cleardevice	graphics.h	installuserfont	graphics.h
clearviewport	graphics.h	line	graphics.h
closegraph	graphics.h	linerel	graphics.h
detectgraph	graphics.h	lineto	graphics.h
drawpoly	graphics.h	moverel	graphics.h
ellipse	graphics.h	moveto	graphics.h
fillellipse	graphics.h	outtext	graphics.h
fillpoly	graphics.h	outtextxy	graphics.h
floodfill	graphics.h	pieslice	graphics.h
getarccoords	graphics.h	putimage	graphics.h
getaspectratio	graphics.h	putpixel	graphics.h
getbkcolor	graphics.h	rectangle	graphics.h
getcolor	graphics.h	registerbgidriver	graphics.h
graphresult	graphics.h	registerbgifont	graphics.h
getdrivername	graphics.h	restorecrtmode	graphics.h

getfillpattern	graphics.h	sector	graphics.h
getfillsettings	graphics.h	setactivepage	graphics.h
getgraphmode	graphics.h	setallpalette	graphics.h
getimage	graphics.h	setaspectratio	graphics.h
getlinesettings	graphics.h	setbcolor	graphics.h
getmaxcolor	graphics.h	setcolor	graphics.h
getmaxmode	graphics.h	setfillpattern	graphics.h
getmaxx	graphics.h	setfillstyle	graphics.h
getmaxy	graphics.h	setgraphbufsize	graphics.h
getmodename	graphics.h	setgraphmode	graphics.h
getmoderange	graphics.h	setlinestyle	graphics.h
getpalette	graphics.h	setpalette	graphics.h
getpalettesize	graphics.h	setrgbpalette	graphics.h
getpixel	graphics.h	settextjustify	graphics.h
gettextsettings	graphics.h	settextstyle	graphics.h
getviewsettings	graphics.h	setusercharsize	graphics.h
getx	graphics.h	setviewport	graphics.h
gety	graphics.h	setvisualpage	graphics.h
graphdefaults	graphics.h	setwritemode	graphics.h
grapherrormsg	graphics.h	textheight	graphics.h
_graphfreemem	graphics.h	textwidth	graphics.h
_graphgetmem	graphics.h	setcursortype	conio.h

A.2.6 Input/Output rutinok

Ezek a rutinok biztosítják az állományszintű és a DOS - szintű I/O lehetőségeket.

access	io.h	getpass	conio.h
cgets	conio.h	gets	stdio.h
chmod	io.h	getw	stdio.h
_chmod	io.h	ioctl	io.h
chsize	io.h	isatty	io.h
clearerr	stdio.h	kbhit	conio.h
close	io.h	lock	io.h
_close	io.h	lseek	io.h
cprintf	conio.h	open	io.h
cputs	conio.h	_open	io.h
creat	io.h	perror	stdio.h
_creat	io.h	printf	stdio.h
creatnew	io.h	putc	stdio.h
creattemp	io.h	putch	conio.h

cscanf	conio.h	putchar	stdio.h
dup	io.h	puts	stdio.h
dup2	io.h	putw	stdio.h
eof	io.h	read	io.h
fclose	stdio.h	_read	io.h
fcloseall	stdio.h	remove	stdio.h
fdopen	stdio.h	rename	stdio.h
feof	stdio.h	rewind	stdio.h
ferror	stdio.h	scanf	stdio.h
fflush	stdio.h	setbuf	stdio.h
fgetc	stdio.h	setcursortype	conio.h
fgetchar	stdio.h	setftime	io.h
fgetpos	stdio.h	setmode	io.h
fgets	stdio.h	setvbuf	stdio.h
filelength	io.h	sopen	io.h
fileno	stdio.h	sprintf	stdio.h
flushall	stdio.h	sscanf	stdio.h
fopen	stdio.h	stat	sys\stat.h
fprintf	stdio.h	sterror	stdio.h
fputc	stdio.h	_sterror	string.h,stdio.h
fputchar	stdio.h	tell	io.h
fputs	stdio.h	tmpfile	stdio.h
fread	stdio.h	tmpnam	stdio.h
freopen	stdio.h	ungetc	stdio.h
fscanf	stdio.h	ungetch	conio.h
fseek	stdio.h	unlock	io.h
fsetpos	stdio.h	vfprintf	stdio.h
fstat	sys\stat.h	vfscanf	stdio.h
ftell	stdio.h	vprintf	stdio.h
fwrite	stdio.h	vscanf	stdio.h
getc	stdio.h	vsprintf	stdio.h
getch	conio.h	vsscanf	io.h
getchar	stdio.h	write	io.h
getche	conio.h	_write	io.h
getftime	io.h		

A.2.7 Interface rutinok (DOS, 8086, BIOS)

Ezek a rutinok segítik a DOS, BIOS és egyéb gépfüggő lehetőségek kihasználását.

absread	dos.h	harderr	dos.h
abswrite	dos.h	hardresume	dos.h

bdos	dos.h	hardretn	dos.h
bdosptr	dos.h	inport	dos.h
bioscom	bios.h	inportb	dos.h
biosdisk	bios.h	int86	dos.h
biosequip	bios.h	int86x	dos.h
bioskey	bios.h	intdos	dos.h
boismemory	bios.h	intdosx	dos.h
biosprint	bios.h	intr	dos.h
biostime	bios.h	keep	dos.h
country	dos.h	MK_FP	dos.h
ctrlbrk	dos.h	outport	dos.h
disable	dos.h	outportb	dos.h
dosexterr	dos.h	parsfnm	dos.h
enable	dos.h	peek	dos.h
FP_OFF	dos.h	peekb	dos.h
FP_SEG	dos.h	poke	dos.h
freemem	dos.h	pokeb	dos.h
geninterrupt	dos.h	randbrd	dos.h
getcbrk	dos.h	randbwr	dos.h
getdfree	dos.h	segread	dos.h
getdta	dos.h	setcbrk	dos.h
getfat	dos.h	setdta	dos.h
getfatd	dos.h	setvect	dos.h
getpsp	dos.h	setverify	dos.h
getvect	dos.h	sleep	dos.h
getverify	dos.h	unlink	dos.h

A.2.8 Memóriablokk- és sztringkezelő rutinok

Ezek a rutinok kezelik a sztringeket és memóriablokkokat, mint másolás, összehasonlítás, konvertálás és keresés.

memcopy	mem.h, string.h	sterror	string.h
memchr	mem.h, string.h	stericmp	string.h
memcmp	mem.h, string.h	strlen	string.h
memcpy	mem.h, string.h	strlwr	string.h
memicmp	mem.h, string.h	strncat	string.h
memmove	mem.h, string.h	strncmp	string.h
memset	mem.h, string.h	strncmpi	string.h
movedata	mem.h, string.h	strncpy	string.h
movmem	mem.h, string.h	strnicmp	string.h

setmem	mem.h	strnset	string.h
stpcpy	string.h	strpbrk	string.h
strcat	string.h	strrchr	string.h
strchr	string.h	strrev	string.h
strcmp	string.h	strset	string.h
strcoll	string.h	strspn	string.h
strcpy	string.h	strsrt	string.h
strcspn	string.h	strtok	string.h
strdup	string.h	strupr	string.h
		strxtrm	string.h

A.2.9 Matematikai rutinok

Ezek a rutinok matematikai műveleteket és konverziókat hajtanak végre. A *-gal jelölt függvényeknek komplex változata is létezik, ezek prototípusait a `complex.h` include file tartalmazza.

*abs	stdlib.h	ldiv	math.h
acos	math.h	*log	math.h
asin	math.h	*log10	math.h
atan	math.h	_lrotl	stdlib.h
atan2	math.h	_lrotr	stdlib.h
atof	stdlib.h, math.h	ltoa	stdlib.h
atoi	stdlib.h	matherr	math.h
atol	stdlib.h	_matherr	math.h
cabs	math.h	modf	math.h
ceil	math.h	poly	math.h
_clear87	float.h	*pow	math.h
_control87	float.h	pow10	math.h
cos	math.h	rand	stdlib.h
cosh	math.h	random	stdlib.h
div	math.h	randomize	stdlib.h
ecvt	stdlib.h	_rotl	stdlib.h
exp	math.h	_rotr	stdlib.h
fabs	math.h	*sin	math.h
fcvt	stdlib.h	*sinh	math.h
floor	math.h	*sqrt	math.h

fmod	math.h	srand	stdlib.h
_fpreset	float.h	_status87	float.h
frexp	math.h	strtod	stdlib.h
gcvt	stdlib.h	strtol	stdlib.h
hypot	math.h	strtoul	stdlib.h
itoa	stdlib.h	*tan	math.h
labs	stdlib.h	*tanh	math.h
ldexp	math.h	ultoa	stdlib.h
arg	complex.h	imag	complex.h
bcd	bcd.h	norm	complex.h
complex	complex.h	polar	complex.h
conj	complex.h	real	complex.h

A.2.10 Dinamikus tárkezelő rutinok

Ezek a rutinok a dinamikus memóriakezelést támogatják. A *-gal jelölt függvények az stdlib.h include file-ban is deklarálva vannak.

allocmem	dos.h	farheapwalk	alloc.h
brk	alloc.h	farmalloc	alloc.h
calloc	alloc.h	farrealloc	alloc.h
coreleft	alloc.h, stdlib.h	free	alloc.h, stdlib.h
farcalloc	alloc.h	heapcheck	alloc.h
farcoreleft	alloc.h	heapcheckfree	alloc.h
farfree	alloc.h	heapchecknode	alloc.h
farheapcheck	alloc.h	heapwalk	alloc.h
farheapcheckfree	alloc.h	*malloc	alloc.h
farheapchecknode	alloc.h	*realloc	alloc.h
farheapfillfree	alloc.h	sbrk	alloc.h
		setblock	dos.h

A.2.11 Különleges rutinok

Ezek a rutinok biztosítják a nem helyi (függvényre lokális) goto lehetőségeket és a hang effektusokat.

delay	dos.h	setjmp	setjmp.h
localeconv	locale.h	setlocale	locale.h
longjmp	setjmp.h	sound	dos.h
nosound	dos.h		

A.2.12 Folyamatvezérlő rutinok

Ezek a rutinok hívják és terminálják az új folyamatokat (process) egy másikon belülről.

abort	process.h	getpid	process.h
execl	process.h	raise	signal.h
execle	process.h	signal	signal.h
execlp	process.h	spawnl	process.h
execlpe	process.h	spawnle	process.h
execv	process.h	spawnlp	process.h
execve	process.h	spawnlpe	process.h
execvp	process.h	spawnv	process.h
execvpe	process.h	spawnve	process.h
exit	process.h	spawnvp	process.h
_exit	process.h	spawnvpe	process.h

A.2.13 Szabványos rutinok

Ezek szabványos (általános célú) rutinok.

abort	stdlib.h	labs	stdlib.h
abs	stdlib.h	lfind	stdlib.h
atexit	stdlib.h	lsearch	stdlib.h
atof	stdlib.h	ltoa	stdlib.h
atoi	stdlib.h	malloc	stdlib.h
atol	stdlib.h	putenv	stdlib.h
bsearch	stdlib.h	qsort	stdlib.h
calloc	stdlib.h	rand	stdlib.h
ecvt	stdlib.h	realloc	stdlib.h
exit	stdlib.h	srand	stdlib.h
_exit	stdlib.h	strtod	stdlib.h
fcvt	stdlib.h	strtol	stdlib.h
free	stdlib.h	swab	stdlib.h
gcvt	stdlib.h	system	stdlib.h
getenv	stdlib.h		
itoa	stdlib.h		

A.2.14 Karaktermódú képernyőkezelő rutinok

Ezek a rutinok karakteres (text) üzemmódban használják a képernyőt.

clreol	conio.h	normvideo	conio.h
clrscr	conio.h	puttext	conio.h
delline	conio.h	setcursortype	conio.h
gettext	conio.h	textattr	conio.h
gettextinfo	conio.h	textbackground	conio.h
gotoxy	conio.h	textcolor	conio.h
highvideo	conio.h	textmode	conio.h
insline	conio.h	wherex	conio.h
lowvideo	conio.h	wherey	conio.h
movetext	conio.h	window	conio.h

A.2.15 Idő és dátum rutinok

Ezek az idő konverziós és dátum-manipuláló rutinok.

astime	time.h	mktime	time.h
ctime	time.h	setdate	dos.h
difftime	time.h	settime	dos.h
dostounix	dos.h	stime	time.h
ftime	sys\timeb.h	strftime	time.h
getdate	dos.h	time	time.h
gettime	dos.h	tzset	time.h
gmtime	time.h	unixtodos	dos.h
localtime	time.h		

A.2.16 Változó argumentumlista kezelő rutinok

Ezek a rutinok használhatók a változó számú argumentummal meghívható függvényekben (mint például a vprintf stb.).

va_arg	stdarg.h	va_start	stdarg.h
va_end	stdarg.h		

A.3 Fontosabb könyvtári függvények

Először az IBM-PC, illetve BORLAND C++ specifikus függvényeket ismertetjük, majd az ANSI C, illetve UNIX kompatibilis, illetve az MS-Windows alatt is használható fontosabb függvényeket és makrókat tekintjük át.

A.3.1 Függvények szöveges üzemmódban

`clreol` `conio.h`

Törli az összes karaktert a sor végéig a képernyő ablakban, anélkül, hogy a kurzort elmozgatná.

```
#include <conio.h>
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.

```
#include <conio.h>
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.

```
#include <conio.h>
void delline(void);
```

Megjegyzés: A `delline` törli a kurzort tartalmazó sort és az alatta lévő sorok egy sorral feljebb lépnek a képernyőn. A `delline` az aktuális képernyő ablakon belül is működik.

`gettext` `conio.h`

Szöveget másol a képernyőről memóriaterületre.

```
#include <conio.h>
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 pozíciónak 2 byte felel meg. A *h* sor és *v* oszlop ta1rola1sa1ra

$$bytes = (h \cdot sor) \cdot (w \cdot oszlop) \cdot 2$$

drab byte szükséges. Sikeres végrehajtás esetén a visszatérési érték 1, különben 0.

`gettextinfo` `conio.h`

A text üzemmódról nyújt információt.

```
#include <conio.h>
void gettextinfo(struct text_info *r);
```

Paraméter: Az `*r` struktúra – ez adja meg a text video informa1cio1kat.
Megjegyzés: A `text_info` struktúra a `conio.h`-ban van definiálva:

```
struct text_info
{
  unsigned char winleft;     /*az ablak bal koordinataja  */
  unsigned char wintop;      /*az ablak felso koordinataja*/
  unsigned char winright;    /*az ablak jobb koordinataja */
  unsigned char winbottom;   /*az ablak also koordinataja */
  unsigned char attribute;   /*szoveg attributum          */
  unsigned char normattr;    /*normal attributum          */
  unsigned char currmode;    /*BW40, BW80, C40 vagy C80   */
  unsigned char screenheight;/*also - felso koordinata    */
  unsigned char screenwidth; /*bal - jobb koordinata      */
```

```
  unsigned char curx;          /*aktualis ablak x koordinata*/
  unsigned char cury;          /*aktualis ablak y koordinata*/
}
```

---

gotoxy conio.h

Pozícionálja a kurzort.

```
#include <conio.h>
void gotoxy(int x, int y);
```

Paraméterek: x, y – a kurzor új pozi1cio1ja.
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 karakterszín kiválasztását végzi.

```
#include <conio.h>
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.

```
#include <conio.h>
void insline(void);
```

Megjegyzés: A sorbeszúrás következtében az utolsó sor kilép a képernyőből.

---

lowvideo conio.h

Normál karakterszín kiválasztását végzi.

```
#include <conio.h>
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.

```
#include <conio.h>
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)! Ezt a `movetext(5,15,20,25,30,5);` függvényhívással tehetjük meg.

`normvideo` `conio.h`

Normál intenzitású karakterszín kiválasztását végzi.

```
#include <conio.h>
void normvideo(void);
```

Megjegyzés: Visszatölti azt a szöveg attributumot, amelyet az indításnál használt a program.

`puttext` `conio.h`

Memóriából képernyőre másol.

```
#include <conio.h>
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.

```
#include <conio.h>
void far _setcursortype(int cur_t);
```

Paraméter:

`cur_t` a kurzortípus kivallasztalsa

Megjegyzés: Alábbi kurzortípusok léteznek:

Kurzortípus	Leírás
`_NOCURSOR`	kikapcsolja a kurzort
`_SOLIDCURSOR`	tömör, téglalap alakú kurzor
`_NORMALCURSOR`	normál _ (aláhúzás) típusú kurzor

Csak BORLAND C++-ban létezik.

`textattr` `conio.h`

Beállítja az előtér és az írás színét.

```
#include <conio.h>
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 a írás színe (0-15)
`bbb`	3 bit a háttér színe (0-7)
`B`	villódzó állapot flag-je

`textbackground` `conio.h`

Kiválasztja a háttér színét.

```
#include <conio.h>
void textbackground(int newcolor);
```

Paraméter: `newcolor` – a háttér színe
Megjegyzés: A `newcolor` megadható (0-7) számértékkel vagy szimbolikus konstanssal. A `newcolor` értéke az alábbi lehet:

szimbolikus konstans	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.

```
#include <conio.h>
void textcolor(int newcolor);
```

Paraméter: `newcolor` – a karakter színe (0-15)
Megjegyzés: A színkonstansok megnevezését lásd a `setallpalette` függvény ismertetésénél. 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.

```
#include <conio.h>
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.

```
#include <conio.h>
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.

```
#include <conio.h>
int wherey(void);
```

Megjegyzés: Visszatérési érték 1-től 25, 43 vagy 50 közötti egész szám.

window conio.h

Szöveg-ablakot definiál a képernyőn.

```
#include <conio.h>
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.

A.3.2 Hangeffektusok létrehozása

`sound` `dos.h`

Megszólaltatja a belső hangszórót az adott frekvencián.

```
#include <dos.h>
void sound(unsigned frequency);
```

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.

```
#include <dos.h>
void delay(unsigned msec);
```

Paraméter: `msec` – a késleltetés ideje [ms] egységben

`nosound` `dos.h`

Kikapcsolja a belső hangszórót.

```
#include <dos.h>
void nosound(void);
```

Példa: A következő program 440 Hz-es hangot (normál zenei A) ad 500 ms-ig.

```
#include <dos.h>
main()
{
    sound(440);
    delay(500);
    nosound();
}
```

A.3.3 Függvények grafikus üzemmódban

---

arc graphics.h

---

x,y középpontú körívet rajzol kezdő és végszög között.

```
#include <graphics.h>
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 eljárás teljes kört rajzol. A szögeket 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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
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`	a téglatest mélysége
`topflag`	ha nem nulla, a téglatest teteje zárt, nulla 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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
void far clearviewport(void);
```

Megjegyzés: Törli a aktuális grafikus ablakot és a kurzort a képernyő ablak (0,0) pozíciójába mozgatja.

---

closegraph graphics.h

Lezárja a grafikus üzemmódot.

```
#include <graphics.h>
void far closegraph(void);
```

Megjegyzés: A **closegraph** a grafikus üzemmód inicializálása előtti képernyő üzemmódot állítja vissza.

---

detectgraph graphics.h

Ellenőrzi a hardvert és meghatározza, hogy milyen grafikus meghajtót és módot lehet használni.

```
#include <graphics.h>
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 grafika használata nem lehetséges, ezt a ***graphdriver** paraméter, illetve a **grahpresult** függvény -2-es visszatérési értéke jelzi. A lehetséges grafikus üzemmód kontsansok az alábbiak:

konstansok	numerikus érték
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 érzékeli 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.

```
#include <graphics.h>
void far drawpoly(int numpoints,
                  int far *polypoints);
```

Paraméterek:

numpoints	a koordináták száma
polypoints	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. Ha hiba történik a poligon rajzolása alatt a **graphresult** -6 értékkel tér vissza.

---

ellipse **graphics.h**

---

x,y középpontú ellipszis ívet rajzol kezdő és végszög között.

```
#include <graphics.h>
void far ellipse(int x, int y, int stangle,
                 int endangle, int xradius,
                 int yradius);
```

Paraméterek:

`x,y`	középpont kezdő koordinátái
`stangle`	kezdeti szög
`endangle`	végszög
`xradius`	vizszintes tengely
`yradius`	függőleges tengely

Megjegyzés: A szögek az óramutató járásával ellentétes irányúak. 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.

```
#include <graphics.h>
void far fillellipse(int x, int y, int xradius,
                     int yradius);
```

Paraméterek:

`x, y`	az ellipszis középpontja
`xradius`	a vizszintes tengely
`yradius`	a függőleges tengely

Megjegyzés: Rajzol egy `x, y` középpontú, `xradius` vizszintes és `yradius` függőleges tengelyű ellipszist és befesti az aktuális színnel és mintával.

`fillpoly` `graphics.h`

Rajzol és befest egy poligont.

```
#include <graphics.h>
void far fillpoly(int numpoints,
                  int far *polypoints);
```

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.

```
#include <graphics.h>
var far floodfill(int x, int y, int border);
```

Paraméterek:

`x, y`	a zárt terület egy belső pontjának koordinátái
`border`	szín

`getarccoords` `graphics.h`

Megadja az utoljára rajzolt ív kezdő- és végkoordinátáinak értékét.

```
#include <graphics.h>
void far getarccoords(struct arccoordstype
                      far *arccoords);
```

Megjegyzés: Visszatér az `arccoordtype` struktúra típusú `*arccoords` változóban elhelyezett értékekkel, ahol

```
struct arccoordstype
{
     int x, y;
     int xstart, ystart, xend, yend;
}
```

Itt

`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 vizszintes/függőleges képarányát.

```
#include <graphics.h>
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

A háttér aktuális színét adja vissza.

```
#include <graphics.h>
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 rajzoló színt adja vissza.

```
#include <graphics.h>
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.

---

getdefaultpalette graphics.h

A paletta (színskála) értékeit adja vissza.

```
#include <graphics.h>
struct palettetype *far getdefaultpalette(void);
```

Megjegyzés: A **palettetype** típusú struktúrára 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.

```
#include <graphics.h>
char *far getdrivername(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 vissza.

```
#inlcude <graphics.h>
void far getfillpattern(char far *pattern);
```

Paraméter: **pattern** – egy pointer, amely egy 8 byte-os szekvenciára mutat.

Megjegyzés: A **setfillpattern** által definiált mintát a **getfillpattern** betölti egy 8 byte-os területre, amit a **pattern** címez meg.

---

getfillsettings graphics.h

Információt ad az aktuális festőmintáról és színről.

```
#include <graphics.h>
void far getfillsettings(struct fillsettingstype
                         far *fillinfo);
```

Megjegyzés: A **getfillsettings** betölti a **fillsettingstype** strukturára mutató pointert a **fillinfo** pointer változóba, amely információt ad az aktális kitöltő mintáról és színről. A struktúra a **graphics.h** file-ban az alábbi:

```
struct fillsettingstype
{
   int pattern;   /* az aktualis minta */
   int color;     /* az aktualis szin  */
 }
```

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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
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 jobb alsó sarka
`*bitmap`	a bufferre mutató pointer

Megjegyzés: A `getimage` a képmező megadott területét elmenti a bitmap

pointer által mutatott memória területre.A memória területnek az első két adata a tartomány szélességét és magasságát tartalmazza, ezután következnek a képmező adatai.

`getlinesettings` `graphics.h`

A vonal típusát, mintáját és vastagságát adja vissza.

```
#include <graphics.h>
void far getlinesettings(struct linesettingstype
                         far *lineinfo);
```

Megjegyzés: A `getlinesettings` betölti a `linesettinsgtype` 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	1	normál vonal (1 pixel széles)
THICK_WIDTH	3	vastag vonal (3 pixel széles)

`getmaxcolor` `graphics.h`

Megadja a maximálisan használható színek számát.

```
#include <graphics.h>
int far getmaxcolor(void);
```

Megjegyzés: Például 256K-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-számot adja meg.

```
#include <graphics.h>
int far getmaxmode(void);
```

Megjegyzés: A legkisebb érték 0.

`getmaxx` `graphics.h`

A maximálisan használható x koordináta értéket adja meg.

```
#include <graphics.h>
int far getmaxx(void);
```

Megjegyzés: Például a CGA kártya 320·200 felbontású grafikus üzemmódja esetén a `getmaxx` 319 értékkel tér vissza.

`getmaxy` `graphics.h`

A maximálisan használható y koordináta értéket adja meg.

```
#include <graphics.h>
int far getmaxy(void);
```

Megjegyzés: Például a CGA kártya 320·200 felbontású grafikus üzemmódja esetén a `getmaxy` 199 értékkel tér vissza.

`getmodename` `graphics.h`

A grafikus eszköz meghajtó nevére mutató pointerrel tér vissza.

```
#include <graphics.h>
char *far getmodename(int mode_number);
```

Paraméter:

`mode_number`	grafikus mód száma

`getmoderange` **`graphics.h`**

Megadja a grafikus meghajtó üzemmódjainak tartományát.

```
#include <graphics.h>
void far getmoderange(int graphdriver,
                      int far *lomode,
                      int far *himode);
```

Paraméterek:

`graphdriver`	grafikus meghajtó
`lmode, hmode`	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.

```
#include <graphics.h>
void far getpalette(struct palettetype far *palette);
```

Megjegyzés: A `getpalette` feltölti a `palettetype` típusú `palette` struktúrát az aktuálisan használt paletta jellemzőivel. A `palettetype` a `graphics.h`-ban van definiálva

```
#define MAXCOLORS  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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
unsigned far getpixel(int x, int y);
```

Paraméterek:

x, y	a képpont koordinátái

gettextsettings graphics.h

Információt ad a beállított írásképről.

```
#include <graphics.h>
void far gettextsettings(struct textsettingstype
                         far *texttypeinfo);
```

Megjegyzés: A **gettextsettings** betölti a **textsettingstype** struktúra címét **atexttypeinfo** pointer változóba. Ez a struktúra tájékoztatást ad a használt karakterkészlet típusáról, irányáról, méretéről és beállítási helyzetéről.

```
struct textsettingstype
{
  int font;         /* karakterkeszlet tipusa */
  int direction;    /* irany                  */
  int charsize;     /* meret                  */
  int horiz;        /* vizszintes helyzet     */
  int vert;         /* fuggoleges helyzet     */
}
```

getviewsettings graphics.h

Az aktuális ablak (viewport) adatait adja meg.

```
#include <graphics.h>
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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
int far gety(void);
```

Megjegyzés: A megadott y koordináta relatív érték az ablakhoz képest.

graphdefaults graphics.h

Alapállapotba állítja vissza a grafikus üzemmódot.

```
#include <graphics.h>
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 a default-ra, valamint a festő típusra, mintára és a szöveginformációra.

---

grapherrormsg graphics.h

Visszatér a hibaüzenetet tartalmazó sztringre mutató pointerrel.

```
#include <graphics.h>
char *far grapherrormsg(int errorcode);
```

Megjegyzés: A hibakódot a **graphresult** függvény szolgáltatja.

_graphfreemem graphics.h

A grafikus memória felszabadítása.

```
#include <graphics.h>
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. Csak BORLAND C++-ban létezik.

_graphgetmem graphics.h

A grafikus memória lefoglalása.

```
#include <graphics.h>
void far *far  _graphgetmem(unsigned size);
```

Paraméter:

`size` a felszabadítandó memória melrete

Megjegyzés: A grafikus könyvtár (nem a felhasználói program) hívja a `_graphgetmem` függggvényt, 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 hívja. Csak BORLAND C++-ban létezik.

`graphresult` `graphics.h`

Az utoljára végrehajtott grafikus művelet hibakódját adja meg.

```
#include <graphics.h>
int far graphresult(void);
```

Megjegyzés: A hibakód táblázata a 4.1.2 szakaszban található.

`imagesize` `graphics.h`

Adott téglalap alakú tartomány méretét adja meg byte-okban.

```
#include <graphics.)
unsigned far imagesize(int left, int top,
                       int right, int bottom);
```

Paraméterek:

`left, top`	a téglalap bal 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.

```
#include <graphics.h>
void far initgraph(int  far *graphdriver,
                   int  far *graphmode,
                   char far *pathtodriver);
```

Paraméterek:

graphdriver	a grafikus kártya típusa
graphmode	grafikus mód
pathtodiver	az aktuális **.bgi** file-t 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** file az aktív DOS könyvtárban van, akkor a **pathtodriver** paraméter értéke "" (üres sztring) lehet.

---

installuserdriver graphics.h

A felhasználó által írt BGI meghajtó installálása a grafikus rendszerbe.

```
#include <graphics.h>
int far installuserdriver(char far *name,
                          int huge (*detect)(void));
```

Paraméterek:

name	az új meghajtó (**.bgi**) file 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.

```
#include <graphics.h>
int far installuser(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, ilnyenkor a függvény a **grError** (-11) értékkel tér vissza.

---

line **graphics.h**

Egy egyenest rajzol két adott pont között.

```
#include <graphics.h>
void far line(int x1, int y1, int x2, int y2);
```

Paraméterek:

x1, y1	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 plot-pozíciótól relatív koordinátákkal megadott pontig.

```
#include <graphics.h>
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 plot-pozíciótól az abszolút koordinátákkal adott pontig.

```
#include <graphics.h>
void far lineto(int x, int y);
```

Paraméterek:

`x, y` az egyenes velgpontja

Megjegyzés: Az egyenest a korábban definiált színnel, vonaltípussal és vonalvastagsággal rajzolja meg.

---

moverel graphics.h

Az aktuális plot-pozíciót áthelyezi a relatív koordinátákkal adott helyre.

```
#include <graphics.h>
void far moverel(int dx, int dy);
```

Paraméterek:

`dx` távolság x irányban
`dy` távolság y irányban

---

moveto graphics.h

A plot-pozíciót az abszolút koordinátákkal adott pontba helyezi át.

```
#include <graphics.h>
void far moveto(int x, int y);
```

Paraméterek:

`x, y` az új plot-pozíció abszolút koordinátái

---

outtext graphics.h

Az aktuális plot-pozíciótól kezdve szöveget ír ki.

```
#include <graphics.h>
void far outtext(char far *textstring);
```

Paraméter:

`textstring` a kiírandó szövegre mutaó pointer

Megjegyzés: A szöveget a korábban beállított betűtípussal, méretben, valamint a kijelölt irányban (vízszintes, függőleges) írja ki a plot-pozíciótól kezdve.

outtextxy **graphics.h**

Szöveget ír ki a megadott (x,y) ponttól kezdve.

```
#include <graphics.h>
void far outtextxy(int x, int y,
                   char far *textstring);
```

Paraméterek:

x, y	az adott pont
textstring	a kiirandó 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 befest.

```
#include <graphics.h>
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 az óramutató járásával ellentéstes 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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
void far putpixel(int x, int y, int color);
```

Paraméterek:

`x, y`	a pont koordinátái
`color`	a pont szine

Megjegyzés: Az (`x, y`) pontban a képpontot az adott színnel rajzolja.

`rectangle` `graphics.h`

Téglalapot rajzol.

```
#include <graphics.h>
void far rectangle(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 téglalapot az aktuális színnel és vonaltípussal rajzolja.

restorecrtmode graphics.h

Visszaállítja azt a képernyő üzemmódot, amelyik az **initgraph** aktiválása előtt volt érvényben.

```
#include <graphics.h>
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 text módokat akarunk váltani.

sector graphics.h

Egy ellipszis ívet rajzol és befest.

```
#include <graphics.h>
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	vizszintes tengely
yradius	függőleges tengely

setactivepage graphics.h

Új lapot nyit meg a grafikus output számára.

```
#include <graphics>
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.

```
#include <graphics.h>
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];
}
```

Lehetséges szinek:

CGA		EGA/VGA	
név	szín	név	szín
BLACK	0	EGA_BLACK	0
BLUE	1	EGA_BLUE	1
GREEN	2	EGA_GREEN	2
CYAN	3	EGA_CYAN	3
RED	4	EGA_RED	4
MAGENTA	5	EGA_MAGENTA	5
BROWN	6	EGA_LIGHTGRAY	7
LIGHTGRAY	7	EGA_BROWN	20
DARKGRAY	8	EGA_DARKGRAY	56
LIGHTBLUE	9	EGA_LIGHTBLUE	57
LIGHTGREEN	10	EGA_LIGHTGREEN	58
LIGHTCYAN	11	EGA_LIGHTCYAN	59
LIGHTRED	12	EGA_LIGHTRED	60
LIGHTMAGENTA	13	EGA_LIGHTMAGENTA	61
YELLOW	14	EGA_YELLOW	62
WHITE	15	EGA_WHITE	63

A tömbben a szín helyén -1 van, ott a paletta színe nem változik.

---

setaspectratio **graphics.h**

Változtatja a figyelembe veendő vizszintes/függőleges képarányt.

```
#include <graphics.h>
void far setaspectratio(int xasp, int yasp);
```

Paraméterek:

`xasp`, `yasp` aránytényezők

Megjegyzés: Ha a beépített képaránnyal egy kör torzult, akkor a hibát szoftver úton kiküszöbölhetjük, ha az arányokat változtatjuk.

---

setbkcolor **graphics.lib**

Beállítja a háttér színét.

```
#include <graphics.h>
void far setbkcolor(int color);
```

Paraméter:

`color` egy szín a palettából, amely lehet egy szám (0–15), vagy a szín szimbólikus neve

Megjegyzés: A `color` paraméterrel az alábbi módon állíthatjuk be kékre a háttér szinét

```
setbkcolor(BLUE);
```

A háttér színként az alábbiakat használhatjuk:

Szám	Név	Szám	Név
0	BLACK	8	DARKGRAY
1	BLUE	9	LIGHTBLUE
2	GREEN	10	LIGHTGREEN
3	CYAN	11	LIGHTCYAN
4	RED	12	LIGHTRED
5	MAGENTA	13	LIGHTMAGENTA
6	BROWN	14	YELLOW
7	LIGHTGRAY	15	WHITE

setcolor graphics.lib

Beállítja a rajzolás színét.

```
#include <graphics.h>
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

Pa-	Pixel szín-értéke			mód
letta	1	2	3	
0	CGA_LIGHTGREEN	CGA_LIGHTRED	CGA_YELLOW	CGAC0
1	CGA_LIGHTCYAN	CGA_LIGHTMAGENTA	CGA_WHITE	CGAC1
2	CGA_GREEN	CGA_RED	CGA_BROWN	CGAC2
3	CGA_CYAN	CGA_MAGENTA	CGA_LIGHTGRAY	CGAC3

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 **CGAC0** 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**)

```
#include <graphics.h>
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 a festőmintát és a színt.

```
#include <graphics.h>
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.

```
#include <graphics.h>
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.

```
#include <graphics.h>
void far setgraphmode(int mode);
```

Paraméter:

mode a beépített grafikus kártya érvényes mo1dja

---

setlinestyle **graphics.h**

Beállítja a vonal típusát és vastagságát.

```
#include <graphics.h>
void far setlinestyle(int linestyle,
                      unsigned upattern,
                      int thickness);
```

Paraméterek:

linestyle	vonaltípus
upattern	vonaltípus minta
thickness	vonalvastagság

Megjegyzés: A **linesettingstype** struktúra a **graphics.h** file-ban 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.

```
#include <graphics.h>
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.

settextjustify graphics.h

Szöveg helyzetének beállítása az **outtext** és az **outtextxy** eljárások számára.

```
#include <graphics.h>
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, vert
RIGHT_TEXT	2	jobbra	horiz
BOTTOM_TEXT	0	aljára	vert
TOP_TEXT	2	tetjére	vert

settextstyle graphics.h

Beállítja az aktuális karakterkészlet típusát és méretét.

```
#include <graphics.h>
void far settextstyle(int font, int direction,
                      int charsize);
```

Paraméterek:

`font`	a karakterészlet neve
`direction`	az írás iránya
`charsize`	karakterek mérete

Megjegyzés: A `font` paraméter az alábbi értékeket veheti fel

név	érték	leírás
`DEFAULT_FONT`	0	$8 \cdot 8$ pixeles karakterek
`TRIPLEX_FONT`	1	3 vonalas karakterek
`SMALL_FONT`	2	kisméretű karakterek
`SANS_SERIF_FONT`	3	talp nélküli karakterek
`GOETHIC_FONT`	4	gótikus karakterek

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:

1	$8 \cdot 8$ bitminájú lesz a karakter
2	$16 \cdot 16$ bitminájú lesz a karakter
...	
10	maximum
0	vagy a beépített 4 -es faktor, vagy a felhasználó által definiált méretű

`setusercharsize` `graphics.h`

A karakter szélességének és magasságának változtatása.

```
#include <graphics.h>
void far setusercharsize(int multx, int divx,
                         int multy, int divy);
```

Paraméterek:

`multx, divx`	`multx / divx` szorzódik a beépített szélességgel
`multy, divy`	`multy / divy` szorzódik a beépített magassággal

`setviewport` `graphics.h`

Ablakot jelöl ki a grafikus képernyőn.

```
#include <graphics.h>
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.

```
#include <graphics.h>
void far setvisualpage(int page);
```

Paraméter:

`page` a lap szalma

Megjegyzés: Több lap használata csak EGA (256K), VGA és Hercules grafikus kártya esetén lehetséges.

`setwritemode` `graphics.h`

Beállítja az írásmódot a vonalrajzolás számára.

```
#include <graphics.h>
void far setwritemode(int mode);
```

Paraméter:

mode kétfajta lehet, az alábbi konstansok közül választhatunk:

Szimbólum	érték	leírás
COPY_PUT	0	másolás
XOR_PUT	1	kizáró vagy

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.

```
#include <graphics.h>
int far textheight(char far *textstring);
```

Paraméter:

textstring szövegre mutató pointer

---

textwidth graphics.h

Visszatér a szöveg képpontokban mért szélességével.

```
#include <graphics.h>
int far textwidth(char far *textstring);
```

Paraméter:

textstring szövegre mutató pointer

A.3.4 Általános könyvtári függvények

A következőkben a BORLAND C++ legfontosabb, általános célú könyvtári függvényeit ismertetjük. Mindegyik függvény esetében utalunk a portabilitási lehetőségekre (ANSI C, és/vagy UNIX kompatibilitás; DOS vagy BORLAND C++ specifikus; csak C++-ban hozzáférhető, stb), illetve az MS-Windows alkalmazói programokban való felhasználási lehetőségre.

abort — **stdlib.h, process.h**

Abnormálisan terminál az egy függvényt.

```
void abort(void);
```

Megjegyzés: Az *Abnormal program termination* szöveget írja ki a standard hiba-periférián (`stderr`) és aktiválja 3-as kóddal az `_exit` fu2ggve1nyt.

Visszatérési érték: 3-as kóddal tér vissza a hívó programba vagy a DOS-ba.

Portabilitás: ANSI C és UNIX kompatibilis.

acos — **math.h, stdlib.h, complex.h**

Arkusz koszinus értéket számol.

Valós verzió:
```
#include <math.h>
double acos(double x);
```

Komplex verzió:
```
#include <complex.h>
double acos(complex x);
```

Megjegyzés: A `double` argumentumnak -1 és 1 között kell lenni. Hibás argumentum esetén 0-val tér vissza, beállítja az `errno` értékét az `EDOM` *Domain error* hibajelzésre. A komplex inverz koszinusz alakja:

$$\arccos(z) = -i \cdot \log(z + i \cdot \sqrt{1 - z^2})$$

A függvény komplex verziója csak C++-ban használható.

Visszatérési érték: $0 \div \pi$ közötti értékkel tér vissza.

Portabilitás: Az `acos` valós változata ANSI C és UNIX kompatibilis. A függvény komplex változata BORLAND C++ specifikus.

asctime — **time.h**

A dátumot és az időt ASCII karakterlánccá konvertálja.

```
#include <time.h>
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 Aug 16 02:05:45 1989\n\0
```

Visszatérési érték: A karakterláncra mutató pointer. A sztringet a következő `asctime()` vagy `ctime()` hívás felülírja.

Portabilitás: ANSI C és UNIX kompatibilis.

`asin` `math.h, complex.h`

Arkusz szinuszt számol.

Valós verzió:	*Komplex verzió:*
`#include <math.h>`	`#include <complex.h>`
`double asin(double x);`	`double asin(complex x);`

Megjegyzés: A `double` argumentumnak -1 és 1 között kell lenni.Hibás argumentum esetén 0-val tér vissza és beállítja `errno` értékét az `EDOM Domain error` hibajelzésre. A komplex inverz szinus alakja:

$$\arcsin(z) = -i \cdot \log(i \cdot z + \sqrt{1 - z^2})$$

Visszatérési érték: $-\frac{\pi}{2} \div \frac{\pi}{2}$ közötti értékkel tér vissza.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis. A függvény komplex verziója csak BORLAND C++-ban használható.

`assert` `assert.h`

Teszteli a feltételt és szükség esetén abortál.

```
#include <assert.h>
#include <stdio.h>
void assert(int condition);
```

Megjegyzés: Ha a `condition` hamis (0), akkor az `assert` a következő üzenetet írja ki a `stderr` folyamba (a terminálra): *Assertion failed:* `condition`, file `forrásfile`, line `sor száma` és az `abort` hívása révén terminálja a programot. Ha az `assert.h` include file beépítése előtt definiáljuk az `NDEBUG` szimbólumot tetszőleges értékkel, akkor az `assert` makró hatástalan lesz.

Portabilitás: ANSI C és UNIX kompatibilis.

atan — math.h, complex.h

Arkusz tangest számít.

Valós verzió:

```
#include <math.h>
double atan(double x);
```

Komplex verzió:

```
#include <complex.h>
double atan(complex 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: $-\frac{\pi}{2} \div \frac{\pi}{2}$ közötti értékkel tér vissza.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatilis. A komplex verziója csak BORLAND C++-ban használható.

atan2 — math.h

y/x arkusz tangensét számítja.

```
#include <math.h>
double atan2(double y, double x);
```

Visszatérési érték: $-\frac{\pi}{2} \div \frac{\pi}{2}$ közötti értékkel tér vissza.

Portabilitás: ANSI C és UNIX kompatibilis.

atof — math.h, stdlib.h

Sztringet konvertál lebegőpontos számmá.

```
#include <math.h>
double atof(const char *s);
```

Visszatérési érték: Az input sztringnek megfelelő lebegőpontos érték. Ha a konvertálás sikertelen, a visszatérési érték 0.

Portabilitás: ANSI C és UNIX kompatibilis.

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: ANSI C és UNIX kompatibilis.

---

atol **stdlib.h**

Sztringet konvertál hosszú egész számmá.

```
#include <stdlib.h>
long atol(const char *s);
```

Visszatérési érték: Az input sztring konvertált értéke. Ha a konvertálás sikertelen, a visszatérési érték `0L`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

bcd **bcd.h**

Számot konvertál binárisan kódolt decimálisba (BCD).

```
#include <bcd.h>
bcd bcd(int x);
bcd bcd(double x);
bcd bcd(double x, int decimals);
```

Megjegyzés: Az összes aritmetikai művelet működik BCD számokkal. A BCD számok 17 jegyre pontosak, abszolút tartományuk 10^{-125} és 10^{125} között van. A **decimals** argumentum opcionális, amellyel megadható, hogy mennyi decimális jegy legyen a tizedes pont után.

Portabilitás: BORLAND C++ specifikus.

---

bdos **dos.h**

A DOS rendszer közvetlen hívását végzi.

```
int bdos(int dosfun,
         unsigned dosdx,
         unsigned dosal);
```

Megjegyzés: *dosfun* az *MS-DOS Programmer's 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 specifikus.

`bsearch` `stdlib.h`

Bináris keresés egy rendezett tömbben.

```
#include <stdlib.h>
void *bsearch(const void *key, const void *base,
      size_t nelem, size_t width,
      int(*fcomp)(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 `NULL`, ha nem talált.

Portabilitás: ANSI C és UNIX kompatibilis.

`cabs` `math.h`

Komplex szám abszolút értéke.

```
#include <math.h>
double cabs(struct complex z);
```

Megjegyzés: A struktúra az alábbi:

```
struct complex
{
  double x, y;
}
```

az `x` a valós, az `y` a képzetes rész.

Visszatérési érték: A `z` komplex szám abszolút értéke vagy `HUGE_VAL` túlcsordulás esetén.

Portabilitás: UNIX kompatibilis.

`calloc` `stdlib.h, alloc.h`

Memóriaterületet foglal le (allokál).

```
#include <stdlib.h>
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: ANSI C és UNIX kompatibilis.

`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 file név nem létezik.

Portabilitás: UNIX kompatibilis.

`clock` `time.h`

Processzor idő lekérdezése.

```
#include <time.h>
clock_t clock(void);
```

Megjegyzés: A `clock` segítségével két esemény közötti időintervallum meghatározható. 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: ANSI C kompatibilis.

`close` `io.h`

Lezár egy file-t.

```
int close(int handle);
```

Megjegyzés: A *handle* file-leíróval azonosított file-t zárja le.

Visszatérési érték: Hiba esetén az `errno` változót beállítja a `EBADF` értékre (rossz file azonosító szám).

Portabilitás: UNIX kompatibilis.

`complex` `complex.h`

Komplex számot hoz létre.

```
#include <complex.h>
complex complex(double real, double imag);
```

Megjegyzés: A megadott valós és képzetes részből komplex számot hoz létre. Ha a képzetes rész 0, akkor az `imag` megadása elhagyható. A C++-ban a `complex` osztály számára a `complex` függvény a konstruktor. A C++ használata szükséges a komplex aritmetikához. Ha nem használjuk a C++-t, akkor csak a `struct complex` és a `cabs` használható, mindkettő a `math.h`-ban van deklarálva. A`complex.h` lehetővé teszi a `+`, `-`, `*`, `/`, `+=`, `-=`, `*=`, `/=`,

==, != operátorok használatát. Szabadon keverhetők a komplex műveletekben az egész, a duplapontosságú és más numerikus típusok. Ugyanúgy használható a << és >> az inputra és outputra, mint más típusoknál.

Visszatérési érték: A komplex szám az adott valós és képzetes részeivel.

Portabilitás: csak C++-ban használható.

`conj` `complex.h`

A komplex szám konjugáltját képezi.

```
#include <complex.h>
complex conj(complex z);
```

Megjegyzés: A `conj(z)` azonos a `complex(real(z),-imag(z))` hívásával.

Visszatérési érték: A komplex szám konjugáltja.

Portabilitás: Csak C++-ban használható.

`cos` `math.h, complex.h`

Koszinusz értéket számol.

Valós verzió:
```
#include <math.h>
double cos(double x);
```

Komplex verzió:
```
#include <complex.h>
double cos(complex x);
```

Megjegyzés: A komplex koszinusz definiciója:

$$\cos(z) = \frac{\exp(i \cdot z) + \exp(-i \cdot z)}{2}$$

Megjegyzés: A függvény komplex verziója csak C++-ban használható.

Visszatérési érték: A radiánban megadott szög koszinusza.

Portabilitás: A függvény valós verziója ANSI C és UNIX kompatibilis. Komplex verziója csak C++-ban használható.

`cosh` `math.h, complex.h`

Koszinusz hiberbolikusz értéket számol.

Valós verzió:	*Komplex verzió:*
`#include <math.h>`	`#include <complex.h>`
`double cosh(double x);`	`double cosh(complex x);`

Megjegyzés: cosh számítása

$$\cosh(x) = \frac{\exp(x) + \exp(-x)}{2}$$

A komplex `cosh` számítása

$$\cosh(z) = \frac{\exp(z) + \exp(-z)}{2}$$

Visszatérési érték: Az argumentum koszinus hiperbolikusza.

Portabilitás: A függvény valós verziója ANSI C és UNIX kompatibilis. A komplex verzió csak C++-ban használható.

`creat` `io.h`

Új file-t hoz létre, vagy felülír egy létező file-t.

```
#include <sys\stat.h>
int creat(const char *path, int amode);
```

Megjegyzés: Létrehozza vagy felülírja a `path`-ban megadott nevű file-t `amode` hozzáférési móddal.

`amode` változó értéke	hozzáférési mód
`S_IWRITE`	csak írásra (DOS alatt olvasást is enged)
`S_IREAD`	csak olvasásra
`S_IREAD | S_IWRITE`	írásra és olvasásra

Visszatérési érték: A file-leíró, vagy hiba esetén -1.

Portabilitás: UNIX kompatibilis.

`ctime` `time.h`

A dátumot és az időt ASCII karakterlánccá konvertálja.

```
#include <time.h>
char *ctime(const time_t *time);
```

Megjegyzés: `*time` egy megelező `time()` hívással állítható be. A dátumot és az időt az alábbi formájú, újsor és `EOS` karakterrel lezárt 26 karakteres sztringben szolgáltatja:

```
Sun Aug 16 02:05:45 1989\n\0
```

Visszatérési érték: A karakterláncra mutató pointer. A sztringet a következő `asctime()`, vagy `ctime()` hívás felülírja.

Portabilitás: ANSI C és UNIX kompatibilis.

---

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: UNIX kompatibilis.

---

eof **io.h**

File-vég ellenőrzése.

```
int eof(int handle);
```

Visszatérési érték: Ha az aktuális pozíció a file-vég, az `eof` visszatérési értéke 1, különben 0.

Portabilitás: DOS specifikus.

`exec...` `process.h`

Programokat tölt be és futtat. *Megjegyzés:* Az `exec...` függvénycsalád egyes tagjai megegyeznek a `spawn...` család megfelelő tagjaival, ha az ott megadott `mode` értéke `P_OVERLAY`. Ezért az `exec...` családban nincs `mode` paraméter, egyebekben a paraméterezés ugyanaz.

Portabilitás: A BORLAND C++ implementáció DOS specifikus, de a UNIX alatt hasonló függvénycsalád létezik.

`exit` `process.h, stdlib.h`

Terminálja a programot és visszaadja a vezérlést a DOS-nak vagy a hívó programnak.

```
void exit(int status);
```

Megjegyzés: A terminálás előtt az összes file-t lezárja. Nem tér vissza. A `status` a visszaadott státuszkód, használható szimbólumok:

`EXIT_SUCCESS`	normál program terminálás
`EXIT_FAILURE`	terminálás hibával.

Portabilitás: ANSI C és UNIX kompatibilis.

`_exit` `process.h, stdlib.h`

Terminálja a programot.

```
void _exit(int status);
```

Megjegyzés: Terminálja a végrehajtást a file-ok lezárása nélkül.

Portabilitás: UNIX kompatibilis.

`exp` `math.h, complex.h`

---

e^x értéket számol.

Valós verzió:

```
#include <math.h>
double exp(double x);
```

Komplex verzió:

```
#include <complex.h>
complex exp(complex.h);
```

Megjegyzés: A komplex exponenciális függvény

$$\exp(x + i \cdot y) = \exp(x) \cdot (\cos(y) + i \cdot \sin(y))$$

Visszatérési érték: e^x.

Portabilitás: A valós változat ANSI C és UNIX kompatibilis, a komplex változat csak C++-ban hozzáférhető.

---

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 *small* és *medium* 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 specifikus.

---

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 specifikus.

---

farfree **alloc.h**

Felszabadít egy memóriablokkot.

```
void farfree(void far *block);
```

Megjegyzés: `block` egy megelőző `farmalloc()`, `farcalloc()` illetve `farrealloc()` hívásból származhat.

Portabilitás: DOS specifikus.

`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()` stb. 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 specifikus.

`fclose` `stdio.h`

File-t zár le.

```
#include <stdio.h>
int fclose(FILE *stream);
```

Megjegyzés: A `stream` file-mutatóval azonosított file-t zárja le.

Visszatérési érték: A visszatérési érték sikeres lezárás esetén 0, különben hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis.

`fcvt` `stdlib.h`

Lebegőpontos számot sztringgé alakít.

```
#include <stdlib.h>
char *fcvt(double value,
           int ndig,
           int *dec,
           int *sign);
```

Megjegyzés: A **value** értékét konvertálja **ndig** decimális 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. **fcvt** következő hívása felül fogja írni az új értékkel.

Portabilitás: ANSI C és UNIX kompatibilis.

---

feof **stdio.h**

File-vég pozíció detektálása.

```
#include <stdio.h>
int eof(FILE *stream);
```

Visszatérési érték: A visszatérési érték nem 0, ha a **stream** file-mutatóval azonosított file aktuális pozíciója a file-vég.

Portabilitás: ANSI C és UNIX kompatibilis.

---

ferror **stdio.h**

Átviteli hiba lekérdezése.

```
#include <stdio.h>
int ferror(FILE *stream);
```

Megjegyzés: A **getc** stb. rutinok mind file 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** file-mutatóval azonosított file írása vagy olvasása közben adatátviteli hiba volt.

Portabilitás: ANSI C és UNIX kompatibilis.

---

fgets **stdio.h**

Egy sor beolvasása adott file-ból.

```
#include <stdio.h>
char *fgets(char *s, int n, FILE *stream);
```

Megjegyzés: Az **stream** által azonosított file-ból 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: ANSI C és UNIX kompatibilis.

---

fopen **stdio.h**

Megnyit vagy létrehoz egy file-t folyam jellegű kezelésre.

```
#include <stdio.h>
FILE *fopen(const char *filename, const char *mode);
```

Megjegyzés: Megnyitja vagy létrehozza a **filename** nevű file-t 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:

r	létező file megnyitása csak olvasásra
w	új file létrehozása (vagy létező felülírása) és megnyitása csak írásra
a	létező file megnyitása hozzáfűzésre (append), vagy új file létrehozása csak írásra, ha nem létezik.
r+	létező file megnyitása olvasásra és írásra
w+	új file létrehozása (vagy létező felülírása) és megnyitása olvasásra és írásra
a+	megnyitás olvasásra és hozzáfűzésre. Ha a file 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 file-t kezelni. Ha nem adjuk meg egyiket sem, akkor a file az **_fmode** nevű globális változó által tárolt mód szerint lesz megnyitva. Az **_fmode**-nak értékül az **fcntl.h** include fileban definiált **O_TEXT** vagy **O_BINARY** szimbólumot adhatjuk.

Visszatérési érték: Sikeres megnyitás esetén a file-mutató, hiba esetén a**NULL** pointer.

Portabilitás: ANSI C és UNIX kompatibilis.

---

fprintf **stdio.h**

Formázott kivitel file-ba.

```
#include <stdio.h>
int fprintf(FILE *stream, const char *format, ...);
```

Megjegyzés: Működése megegyezik a **printf** rutinéval, de megadható a kimeneti folyam a **stream** file-mutatóval. További részleteket a **printf** függvénycsalád ismertetésénél, a 2.10.2-es szakaszban találhatunk.

Visszatérési érték: A kiírt byte-ok száma, vagy hiba esetén **EOF**.

Portabilitás: ANSI C és UNIX kompatibilis.

---

fputs **stdio.h**

File-ba kiír egy karakterláncot.

```
#include <stdio.h>
int fputs(const char *s, FILE *stream);
```

Megjegyzés: Kiírja az **s** sztringet a **stream**-mel azonosított file-ba. A záró **EOS** nem kerül kivitelre.

Visszatérési érték: Sikeres végrehajtás esetén az utolsó kiírt karakterrel tér vissza, egyébként a visszatérési érték **EOF**.

Portabilitás: ANSI C és UNIX kompatibilis.

---

fread **stdio.h**

File-ból tömböt olvas.

```
#include <stdio.h>
size_t fread(void *ptr, size_t size, size_t n,
             FILE *stream);
```

Megjegyzés: A **stream**-mel azonosított file-ból beolvas **n** darab **size** méretű adatot a **ptr** által mutatott tömbbe.

Visszatérési érték: A beolvasott adatok (nem a byte-ok) száma. Hiba, vagy file-vég esetén `n`-nél kevesebbet ad vissza.

Portabilitás: ANSI C és UNIX kompatibilis.

`free` **`stdlib.h, alloc.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: ANSI C és UNIX kompatibilis.

`fscanf` **`stdio.h`**

File-ból olvas formátum szerint

```
#include <stdio.h>
int fscanf(FILE *stream, const char *format, ...);
```

Megjegyzés: Működése megegyezik a `scanf` rutinéval, de megadható a bemeneti folyam a `stream` file-mutatóval.

Visszatérési érték: A sikeresen beolvasott mezők száma. File-vég esetén `EOF`-ot ad.

Portabilitás: ANSI C és UNIX kompatibilis.

`fseek` **`stdio.h`**

File aktuális pozíciójának beállítása

```
#include <stdio.h>
int fseek(FILE *stream, long offset, int origin);
```

Megjegyzés: Beállítja a `stream` által azonosított file-ban az aktuális file-pozíciót az `origin`-hez képest `offset` byte-ra. Az `origin` az alábbi értékeket veheti fel:

szimbólum	számérték	`offset` számítása
`SEEK_SET`	0	a file elejétől
`SEEK_CUR`	1	az aktuális file pozíciótól
`SEEK_END`	2	a file végétől

Visszatérési érték: Sikeres végrehajtás esetén 0, különben nem 0.

Portabilitás: ANSI C és UNIX kompatibilis.

`ftell` **`stdio.h`**

Az aktuális file-pozíciót szolgáltatja.

```
#include <stdio.h>
long int ftell(FILE *stream);
```

Megjegyzés: A file-pozíció a file elejétől byte-okban kifejezett távolság,`0L`-ról indul.

Visszatérési érték: Az aktuális file-pozíció értéke sikeres végrehajtás esetén, egyébként `-1L`.

Portabilitás: ANSI C és UNIX kompatibilis.

`fwrite` **`stdio.h`**

File-ba tömböt ír.

```
#include <stdio.h>
size_t fwrite(const void *ptr, size_t size,
              size_t n, FILE *stream);
```

Megjegyzés: A `stream`-mel azonosított file-ba kiír `n` darab `size` méretű adatot a `ptr` mutatta tömbből.

Visszatérési érték: A kiírt adatok (nem a byte-ok) száma. Hiba esetén `n`-nél kevesebbet ad vissza.

Portabilitás: ANSI C és UNIX kompatibilis.

`gcvt` **`stdlib.h`**

Lebegőpontos szám ASCII karakterlánccá konvertálása.

```
#include <dos.h>
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: ANSI C és UNIX kompatibilis.

---

getc **stdio.h**

Egy karaktert olvas a file-ból.

```
#include <stdio.h>
int getc(FILE *stream);
```

Megjegyzés: Beolvassa a file következő karakterét, és egész számmá konvertálja előjelkiterjesztés nélkül. `getc` makróként lett megvalósítva.

Visszatérési érték: A beolvasott karakter, file-vég vagy hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

getch **conio.h**

Karakter olvas a klaviatúráról, képernyőre írás (echo) nélkül.

```
int getch(void);
```

Megjegyzés: A `getch` a `stdin` folyamot (standard input) használja.

Visszatérési érték: a beolvasott karakter.

Portabilitási opciók: DOS specifikus, WINDOWS-ban nem használható.

---

getchar **stdio.h**

Karaktert olvas a standard inputról.

```
#include <stdio.h>
int getchar(void);
```

Megjegyzés: Megegyezik a `getc(stdin)` hívással.

Visszatérési érték: a beolvasott karakter.

Portabilitás: ANSI C és UNIX kompatibilis. WINDOWS-ban nem használható.

---

getche — **conio.h**

Karaktert olvas a klaviatúráról és visszaírja a képernyőre.

```
int getche(void);
```

Visszatérési érték: a beolvasott karakter.

Portabilitás: ANSI C és UNIX kompatibilis. WINDOWS-ban nem használható.

---

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`, file-vég vagy hiba esetén `NULL`.

Portabilitás: ANSI C és UNIX kompatibilis. WINDOWS-ban nem használható.

---

gmtime — **time.h**

A `time_t` típusban megadott dátumot és időt struktúrába bontja.

```
#include <time.h>
struct tm *gmtime(const time_t *time);
```

Megjegyzés: **time** feltöltését a **time** hívásával végezhetjük. A **tm** struktúra definíciója a **time.h** include file-ban található.

Visszatérési érték: A struktúrára mutató pointer. A következő **gmtime()** hívás felülírja az új értékkel.

Portabilitás: DOS specifikus.

---

int86 **dos.h**

Általános 8086 programozott megszakítás (*software interrupt*)

```
#include <dos.h>
int int86(int intno,
          union REGS *inregs,
          union REGS *outregs);
```

Megjegyzés: Az **int86** az **intno** argumentumban specifikált 8086 szoftver interruptot hajtja végre. Mielőtt végrehajtaná a szoftver interruptot, átmásolja a regiszterek értékét az **inregs** paraméterből a regiszterekbe. Az interrupt végrehajtása után az **int86** átmásolja a regiszterek tartalmát az **outregs**-be. A carry flag státuszát a **x.cflag** tartalmazza.

Visszatérési érték: A visszaadott AX regiszterérték.

Portabilitás: A 8086 processzor családra specifikus.

---

int86x **dos.h**

Általános 8086 programozott megszakítás (*software interrupt*)

```
#include <dos.h>
int int86x(int intno,
           union REGS *inregs,
           union REGS *outregs,
           struct SREGS *segregs);
```

Megjegyzés: Az **intno** argumentumban specifikált 8086 szoftware interruptot hívja meg. **inregs** és **outregs** a be-, illetve kimenő regiszterkészlet mutatói (lehetnek azonosak), **segregs** a szegmens regiszterkészlet mutatója. A fenti struktúra definíciója a **dos.h** fejlécfile-ban található.

Visszatérési érték: A visszaadott AX regiszterérték.

Portabilitás: 8086 processzor családra specifikus.

`isalnum` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int isalnum(int c);
```

Visszatérési érték: nem nulla, ha a `c` betű (A-Z, a-z) vagy számjegy (0-9).

Portabilitás: UNIX specifikus.

`isalpha` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int isalpha(int c);
```

Visszatérési érték: nem nulla, ha a `c` betű (A-Z vagy a-z).

Portabilitás: ANSI C és UNIX kompatibilis.

`isascii` `ctype.h`

Karakter osztályozó.

```
#include <ctype>
int isascii(int c);
```

Visszatérési érték: nem nulla, ha a `c` alsó byte-jának értéke 0–127 (`0x00 - 0x7F`).

Portabilitás: UNIX kompatibilis.

`iscntrl` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int iscntr(int c);
```

Visszatérési érték: nem nulla, ha a `c` értéke a `DEL` karakter, vagy vezérlőkarakter (`0x7F`, vagy `0x00 - 0x1F`).

Portabilitás: ANSI C és UNIX kompatibilis.

`isdigit` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int isdigit(int c);
```

Visszatérési érték: nem nulla, ha a `c` értéke számjegy: (0 – 9).

Portabilitás: ANSI C és UNIX kompatibilis.

`islower` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int islower(int c);
```

Visszatérési érték: nem nulla, ha a `c` kisbetű (a – z);

Portabilitás: ANSI C és UNIX kompatibilis.

`isprint` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
int isprint(int c);
```

Visszatérési érték: nem nulla, ha a `c` nyomtatható karakter (`0x20 - 0x7E`).

Portabilitás: ANSI C és UNIX kompatibilis.

`isspace` `ctype.h`

Karakter osztályozó makró.

```
#include <ctype.h>
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`).

Portabilitás: ANSI C és UNIX kompatibilis.

---

isupper ctype.h

Karakter oszályozó makró.

```
#include <ctype.h>
int isupper(int c);
```

Visszatérési érték: nem nulla, ha a `c` nagybetű (A – Z).

Portabilitás: ANSI C és UNIX kompatibilis.

---

isxdigit ctype.h

Karakter osztályozó makró.

```
#include <ctype.h>
int isxdigit(int c);
```

Visszatérési érték: nem nulla, ha a `c` hexadecimális számjegy (0 – 9, A – F, a – f).

Portabilitás: ANSI C és UNIX kompatibilis.

---

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 specifikus.

kbhit **conio.h**

Megvizsgálja, hogy várakozik-e karakter a billentyű-pufferben.

```
int kbhit(void);
```

Visszatérési érték: Nem nulla, ha van beolvasható karakter.

Portabilitási opciók: DOS specifikus, WINDOWS-ban nem használható.

keep **dos.h**

Terminálja a processzt, de rezidensen a tárban hagyja.

```
void keep(unsigned char status, unsigned size);
```

Megjegyzés: **status** státuszkóddal tér vissza a DOS-hoz, és a program **size** paragrafus hosszú részét (**size*16** byte) rezidensen a tárban hagyja.

Visszatérési érték: Nem tér vissza.

Portabilitási opciók: DOS specifikus, WINDOWS-ban nem használható.

lfind **stdlib.h**

Lineáris keresést hajt végre.

```
#include <stdlib.h>
void *lfind(const void *key, const void *base,
            size_t *num, size_t width,
            int (*fcmp)(const void *, const void *));
```

Megjegyzés: A ***key** értéke szerint lineáris keresés történik a **base** tömbben a felhasználó által definiált **fcmp** összehasonlító rutin felhasználásával (lásdd **bsearch**). A tömb ***num** elemből áll, **width** az elemek **sizeof** mérete.

Visszatérési érték: Az első egyező tömbelem címe, **NULL**, ha nincs ilyen.

Portabilitás: DOS specifikus.

log **math.h, complex.h**

Természetes alapú logaritmust számol.

Valós verzió:	*Komplex verzió:*
`#include <math.h>`	`#include <complex.h>`
`double log(double x);`	`double log(complex x);`

Visszatérési érték: `ln(x)`, hibás argumentum esetén `EDOM (Domain error)` hibajelzést ad.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, komplex verziója csak C++-ban használható.

`log10` `math.h, complex.h`

Tizes alapú logaritmust számol.

Valós verzió:	*Komplex verzió:*
`#include <math.h>`	`#include <complex.h>`
`double log10(double x);`	`double log10(complex x);`

Visszatérési érték: `lg(x)`, hibás argumentum esetén `EDOM (Domain error)` hibajelzést ad.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, komplex változata csak C++-ban használható.

`lsearch` `stdlib.h`

Lineáris keresést hajt végre.

```
#include <stdlib.h>
void *lsearch(const void *key, const void *base,
              size_t *num, size_t width,
              int (*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: UNIX kompatibilis.

lseek **io.h**

File-pozíciót állít be alacsony szintű I/O művelethez.

```
#include <io.h>
long lseek(int handle, long offset, int origin);
```

Megjegyzés: Beállítja a **handle** file-leíró által azonosított file-ban az aktuális file-pozíciót az **origin**-hez képest **offset** byte-ra. Az **origin** az alábbi értékeket veheti fel:

szimbólum	számérték	**offset** számítása
SEEK_SET	0	a file elejétől
SEEK_CUR	1	az aktuális file pozíciótól
SEEK_END	2	a file végétől

Visszatérési érték: Sikeres végrehajtás esetén a beállított file-pozíció a file elejétől számítva, egyébként -1L.

Portabilitás: UNIX kompatibilis.

ltoa **stdlib.h**

Hosszú egész számot sztringgé konvertál.

```
#include <stdlib.h>
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 előjelesen konvertál, egyébként előjeltelenül.

Visszatérési érték: a **sztring**-re mutató pointer.

Portabilitás: DOS specifikus.

malloc **stdlib.h, alloc.h**

Dinamikus memóriafoglalás.

```
#include <alloc.h>
void *malloc(size_t size);
```

Megjegyzés: `size` byte-nyi memóriát igényel futás közben.

Visszatérési érték: az újonnan lefoglalt memóriablokkra mutató pointer, ha a kérés teljesíthető, egyébként `NULL`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

matherr — **math.h**

A felhasználó által módosítható matematikai hibakezelő rendszer.

```
#include <math.h>
int matherr(struct exception *e);
```

Megjegyzés: A `matherr` függvény matemetikai jellegű hibák felléptekor (például 0-val osztás, lebegőpontos túl-, illetve alulcsordulás stb.) kerül meghívásra. Ha a felhasználó nem definiálja a fenti függvényt, akkor az alapértelmezés szerinti hibakezelő lép működésbe.

Portabilitás: ANSI C és UNIX kompatibilis.

---

memchr — **string.h, mem.h**

Megkeresi egy adott byte első előfordulási helyét.

```
#include <mem.h>
void *memchr(const void *s, int c, size_t n);
```

Megjegyzés: Az `s` tömbben keresi a `c` karakter első előfordulását. `n` a tömb mérete byte-ban.

Visszatérési érték: A megtalált byte-ra mutató pointer, illetve `NULL`, ha nem talall.

Portabilitás: ANSI C és UNIX kompatibilis.

---

memcmp — **string.h, mem.h**

Két adott méretű blokkot hasonlít össze.

```
#include <mem.h>
int memcmp(const void *s1, const void *s2, size_t n);
```

Megjegyzés: Az `s1` és `s2` tömb első `n` byte-ját hasonlítja össze lexikografikusan.

Visszatérési érték: negatív, ha `s1` < `s2`, nulla, ha `s1` = `s2` és pozitív, ha `s1` > `s2`.

Portabilitás: ANSI C és UNIX kompatibilis.

`memcpy` `string.h, mem.h`

Adott méretű memóriablokkot másol.

```
#include <mem.h>
void *memcpy(void *dest, const void *src, size_t n);
```

Megjegyzés: `n` byte-nyi blokkot másol az `src` területről a `dest` területre. Átfedő területek esetén nem alkalmazható.

Visszatérési érték: `dest`.

Portabilitás: ANSI C és UNIX kompatibilis.

`memset` `string.h, mem.h`

Adott hosszúságú memóriaterület feltöltése.

```
#include <mem.h>
void *memset(void *s, int c, size_t n);
```

Megjegyzés: Az `s` tömb első `n` byte-jába beírja a `c` karaktert.

Visszatérési érték: `s`.
Portabilitás: ANSI C és UNIX kompatibilis.

`open` `io.h`

File-nyitás alacsony szintű írásra/olvasásra.

```
#include <fcntl.h>
#include <sys\stat.h>
int open(const char *name,
         int mode, unsigned attrib);
```

Megjegyzés: Megnyitja a **name** nevű file-t, és előkészíti írásra és/vagy olvasásra a **mode** paraméter szerint. A **mode** az alábbi szimbólumok bináris OR kapcsolatával (|) képezhető. Ebből a csoportból pontosan egy szimbólum szerepelhet:

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 file-pozíció a file végére lesz állítva.
O_CREAT	Ha a file nem létezik, akkor létre kell hozni.
O_EXCL	Ha a file létezik és O_CREAT kérelem volt, hibával tér vissza.
O_TRUNC	Ha a file létezik, akkor levágja 0 hosszúságúra.
O_BINARY	A file-t bináris kezelési módban nyitja meg.
O_TEXT	A file-t 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 file-leíró sikeres végrehajtás esetén, egyébként -1.

Portabilitás: UNIX kompatibilis.

---

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: ANSI C és UNIX kompatibilis.

---

printf **stdio.h**

Formattált kimenet az stdout-ra.

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

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 soronkövetkező argumentum megfelelő feldolgozását és kiíratását írják elő). Részletes ismertetőt találhatunk a `printf` függvényről a 2.10.2-es szakaszban.

Visszatérési érték: A kiírt byte-ok száma, hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis. WINDOWS-ban nem használható.

`putc` `stdio.h`

Karaktert ír ki egy folyam-jellegű file-ba.

```
#include <stdio.h>
int putc(int c, FILE *stream);
```

Megjegyzés: Makró, amely kiírja a `c` karaktert a `stream` file-mutatójú állományba.

Visszatérési érték: A kiírt karakter, hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis.

`putchar` `stdio.h`

Karaktert ír az `stdout` peirfériára.

```
#include <stdio.h>
int putchar(int c);
```

Megjegyzés: Makró, amely megfelel a `putc(c, stdout)` hívásnak.

Visszatérési érték: A kiírt karakter, hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

puts **stdio.h**

Karakterláncot ír ki az **stdout** perifériára.

```
#include <stdio.h>
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: ANSI C és UNIX kompatibilis.

---

qsort **stdlib.h**

Tömb rendezése gyorsrendező (quicksort) algoritmussal.

```
#include <stdlib.h>
void qsort(void *base, size_t nelem, size_t width,
           int (*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.

Portabilitási: ANSI C és UNIX kompatibilis.

rand **stdlib.h**

Véletlenszám-generátor.

```
#include <stdlib.h>
int rand(void);
```

Visszatérési érték: Az álvéletlen szám 0 és **RAND_MAX** ko2zo2tt.

Portabilitás: ANSI C és UNIX kompatibilis.

random **stdlib.h**

Véletlenszám-generátor.

```
#include <stdlib.h>
int random(int num);
```

Visszatérési érték: 0 és (**num-1**) közötti véletlenszám.

Portabilitás: BORLAND C++ specifikus, ez a függvény megtalálható a Turbo Pascal-ban is.

read **io.h**

Alacsony szintű file-olvasás.

```
#include <io.h>
int read(int handle, void *buf, unsigned len);
```

Megjegyzés: **len** byte-ot olvas a **handle**-vel azonosított file-ból **buf**-ba.

Visszatérési érték: A sikeresen beolvasott byte-ok száma. File-vég esetén 0, hiba esetén -1.

Portabilitás: UNIX kompatibilis.

realloc **stdlib.h, alloc.h**

Módosítja a **malloc**, illetve a **calloc** által lefoglalt memóriablokkot.

```
#include <stdlib.h>
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: ANSI C és UNIX kompatibilis.

---

scanf **stdio.h**

Olvassa és formázza az **stdin** bemenetet.

```
#include <stdio.h>
int scanf(const char *format, ...);
```

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 2.10.2-es részben.

Visszatérési érték: A sikeresen beolvasott és eltárolt tételek száma. File-vég olvasása esetén **EOF**-ot ad vissza.

Portabilitás: ANSI C és UNIX kompatibilis.

---

setbuf **stdio.h**

Egy folyamhoz rendelt buffer méretét állítja be.

```
#include <stdio.h>
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: ANSI C és UNIX kompatibilis.

setmode io.h

A file kezelési módját állítja át.

```
#include <fcntl.h>
int setmode(int handle, int mode);
```

Megjegyzés: a **mode** értéke az alábbi lehet:

O_BINARY	bináris
O_TEXT	text (szöveg) típusú.

Visszatérési érték: nulla, ha sikeres a végrehajtás.

Portabilitás: UNIX kompatibilis.

setvbuf stdio.h

A folyam bufferelését írja elő.

```
#include <stdio.h>
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: ANSI C és UNIX kompatibilis.

sin math.h, complex.h

Szinusz értéket számol.

Real verzió:
```
#include <math.h>
double sin(double x);
```
Komplex verzió:
```
#include <complex.h>
double sin(complex x);
```

Visszatérési érték: `x` szinusza.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, a komplex változata csak C++-ban használható.

`sinh` `math.h, complex.h`

Szinusz hiperbolikusz értéket számol.

Real verzió:
```
#include <math.h>
double sinh(double x);
```
Komplex verzió:
```
#include <complex.h>
double sinh(complex x);
```

Visszatérési érték: $\sinh(x)$. *Portabilitás:* A függvény valós változata ANSI C és UNIX kompatibilis, komplex változata csak C++-ban használható.

`spawn...` `process.h`

Ez egy függvénycsalád. Az itt szereplő függvények arra szolgálnak, hogy egy adott program kezdeményezhesse egy másik program (*subprocess*, *child process*) operatív tárba való behívását és futtatását.

```
#include <process.h>
#include <stdio.h>
int spawnl(int mode, char *path,
           char *arg0,
           char *arg1, ...,
           char *argn, NULL);

int spawnle(int mode, char *path,
            char *arg0,
            char *arg1, ...,
            char *argn, NULL,
            char *envp[]);

int spawnlp(int mode, char *path,
            char *arg0,
            char *arg1, ...,
            char *argn, NULL);
```

```
int spawnlpe(int mode, char *path,
             char *arg0,
             char *arg1, ...,
             char *argn, NULL,
             char *envp[]);

int spawnv(int mode, char *path, char * argv[]);

int spawnve(int mode, char *path, char *argv[],
            char *envp[]);

int spawnvp(int mode, char *path, char *argv[]);

int spawnvpe(int mode, char *path, char *argv[],
             char *envp[]);
```

Megjegyzés: A **spawn...** család gyerekfolyamatként (*child process*) betölt és futtat egy adott programot. A **spawn...** függvény nevéhez adott karakterek jelentése:

- **p** a file-t az aktuális DOS könyvtáron kívül a **PATH** környezeti változóban megadott könyvtárakban is keresi,
- **l** **arg0, arg1, ... argn** felsorolásával adjuk át a gyereknek a paramétereket,
- **v** az argumentumokra mutató pointerek az **argv[ ]** tömbben kerülnek átadásra, **NULL**-pointerrel lezárva (akkor célszerű, ha az argumentumok száma futás közben dől el),
- **e** a környezeti változókat tartalmazó pointertömb is átadásra kerül. A tömb utolsó eleme kötelezően **NULL**.

Paraméterek:

mode	a működési módot határozza meg
path	a betöltendő gyerekprogramot tartalmazó file neve

A **mode** paraméter értéke az alábbi lehet:

P_WAIT	a szülőfolyamat felfüggesztve, míg a gyerek fut,
P_NOWAIT	a szülő és a gyerek párhuzamosan futnak (nem alkalmazható DOS alatt)
P_OVERLAY	a gyerekfolyamat felülírja a szülő által elfoglalt memória-területet.

Visszatérési érték: hiba esetén -1, egyébként a gyerekfolyamat által visszaadott státuszkód. Sikeres `P_OVERLAY` esetén nem tér vissza.

Portabilitás: A `spawn` függvények DOS specifikusak. Az `exec` függvénycsalád név-kiterjesztése és paraméterezése olyan, mint a `spawn` függvénycsalád tagjaié, hívásuk szintaktikailag UNIX kompatibilis, végrehajtási módjuk DOS specifikus...

`sprintf` `stdio.h`

Formatált kimenetet ír egy sztringbe.

```
int sprintf(char *buffer, const char *format, ...);
```

Megjegyzés: Megegyezik a `print` függvénnyel, de a kimenetét a `buffer` memóriaterületen helyezi el.

Visszatérési érték: a kiírt byte-ok száma a lezáró `EOS` karakter nélkül.

Portabilitás: ANSI C és UNIX kompatibilis.

`sqrt` `math.h, complex.h`

Négyzetgyököt számol.

Real verzió:

```
#include <math.h>
double sqrt(double x);
```

Komplex verzió:

```
#include <complex.h>
double sqrt(complex x);
```

Visszatérési érték: az `x` négyzetgyöke.

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, a komplex változata csak C++-ban használható.

`srand` `stdlib.h`

Inicializálja a véletlenszám generátort.

```
#include <stdlib.h>
void srand(unsigned seed);
```

Megjegyzés: a **seed** értékével a véletlenszám generátornak új kezdőértéket adhatunk.

Portabilitás: ANSI C és UNIX kompatibilis.

---

sscanf **stdio.h**

Sztringből formattált adatot olvas.

```
#include <stdio.h>
int sscanf(const char *buffer,
           const char *format, ...);
```

Megjegyzés: Megegyezik az **scanf** függvénnyel, de a bemenetét a **buffer** által megadott memóriaterületről veszi.

Visszatérési érték: a sikeresen beolvasott és tárolt mezők száma.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strcat **string.h**

Sztringhez egy másik karakterláncot fűz.

```
#include <string.h}
char *strcat(char *dest, const char *src);
```

Megjegyzés: Az **src** sztringet hozzámásolja a **dest** végéhez.

Visszatérési érték: Az összefűzött sztringre mutató pointer.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strchr **string.h**

Egy sztringben adott karakter első előfordulását keresi.

```
#include <string.h>
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: ANSI C és UNIX kompatibilis.

---

strcmp string.h

Két sztringet hasonlít össze.

```
#include <string.h>
int strcmp(const char *s1, const char *s2);
```

Visszatérési érték: negatív, ha `s1 < s2`, nulla, ha `s1 == s2` és pozitív, ha `s1 > s2`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strcpy string.h

Egy sztringet másikba másol.

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

Megjegyzés: Az `src` sztringet másolja a `dest` sztringbe a lezáró `EOS` karakterrel bezárólag.

Visszatérési érték: `dest`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strdup string.h

Sztringet másol egy dinamikusan foglalt tárterületre.

```
#include <string.h>
char *strdup(const char *s);
```

Visszatérési érték: A másolatra mutató pointer, ill. `NULL`, ha nem sikerült a területfoglalás.

Portabilitás: UNIX kompatibilis.

stricmp **string.h**

Két sztringet hasonlít össze, kis- és nagybetűket azonosnak véve.

```
#include <string.h>
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 specifikus.

strlen **string.h**

A sztring hosszát adja meg.

```
#include <string.h>
size_t strln(const char *s);
```

Visszatérési érték: az `s` sztring hossza a lezáró `EOS` karakter nélkül.

Portabilitás: ANSI C és UNIX kompatibilis.

strlwr **string.h**

Egy sztring nagybetűit kisbetűkre cseréli le.

```
#include <string.h>
char *strlwr(char *s);
```

Visszatérési érték: `s`.

Portabilitás: DOS specifikus.

strncat **string.h**

Maximált hosszúságú karakterláncot fűz egy sztringhez.

```
#include <string.h>
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: ANSI C és UNIX kompatibilis.

---

strncmp string.h

Összehasonlít két maximált hosszúságú karakterláncot.

```
#include <string.h>
int strncmp(const char *s1,
            const char *s2,
            size_t maxlen);
```

Megjegyzés: Megegyezik `strcmp`-rel, de mindkét sztringből maximum `maxlen` karaktert vesz figyelembe.

Visszatérési érték: negatív, ha `s1` < `s2`, nulla, ha `s1 == s2` és pozitív, ha `s1` > `s2`.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strncpy string.h

Maximált hosszú sztring másolása.

```
#include <stdio.h>
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: ANSI C és UNIX kompatibilis.

---

strnset string.h

Egy sztring első n karakterét feltölti adott c értékkel.

```
#include <string.h>
char *strnset(char *s, int c, size_t n);
```

Visszatérési érték: s.

Portabilitás: DOS specifikus.

---

strrchr **string.h**

Egy sztringben adott karakter utolsó előfordulását keresi.

```
#include <string.h>
char *strrchr(const char *s, int c);
```

Visszatérési érték: a c karakter s-beli utolsó előfordulási helyére mutató pointer. Ha c nem található s-ben, NULL értékkel tér vissza.

Portabilitás: ANSI C és UNIX kompatibilis.

---

strrev **string.h**

Helyben megfordít egy sztringet, az EOS karaktert helyén hagyva.

```
#include <string.h>
char *strrev(char *s);
```

Visszatérési érték: Pointer a megfordított sztringre.

Portabilitás: DOS specifikus.

---

strset **string.h**

Egy adott karakterrel feltölt egy sztringet.

```
#include <string.h>
char *strset(char *s, int c);
```

Visszatérési érték: s.

Portabilitás: DOS specifikus.

strstr — **string.h**

Egy sztringben adott részlánc első előfordulását keresi.

```
#include <string.h>
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: ANSI C és UNIX kompatibilis.

strupr — **string.h**

Egy sztring kisbetűit nagybetűkre cseréli le.

```
char *strupr(char *s);
```

Visszatérési érték: **s**.

Portabilitás: DOS specifikus.

system — **stdlib.h, process.h**

Az operációs rendszer egy parancsát hajtja végre.

```
int system(const char *command);
```

Megjegyzés: Behívja DOS **COMMAND.COM** file-t, hogy végrehajtsa a **command** sztringben megadott DOS parancsot.

Visszatérési érték: nulla, ha sikeres a végrehajtás, különben -1.

Portabilitás: ANSI C és UNIX kompatibilis. A WINDOWS-ban nem használható.

tan — **math.h, complex.h**

Tangens értéket számol.

Real verzió:
```
#include <math.h>
double tanh(double x);
```

Komplex verzió:
```
#include <complex.h>
double tanh(complex x);
```

Visszatérési érték: tan(x)

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, a komplex változata csak C++-ban használható.

---

tanh **math.h, complex.h**

Tangens hiperbolikusz értéket számol.

Real verzió:
```
#include <math.h>
double tanh(double x);
```

Komplex verzió:
```
#include <complex.h>
double tanh(complex x);
```

Visszatérési érték: tanh(x)

Portabilitás: A függvény valós változata ANSI C és UNIX kompatibilis, a komplex változata csak C++-ban használható.

---

tell **io.h**

Lekérdezi az aktuális file-pozíciót.

```
#include <io.h>
long tell(int handle);
```

Visszatérési érték: az aktuális file-pozíció, hiba esetén -1.

Portabilitás: UNIX kompatibilis.

---

time **time.h**

Az aktuális időt kérdezi le.

```
#include <time.h>
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: ANSI C és UNIX kompatibilis.

---

toascii **ctype.h**

ASCII karakterré alakít.

```
#include <ctype.h>
int toascii(int c);
```

Megjegyzés: Az alsó 7 bit kivételével törli `c` összes bitjét.

Visszatérési érték: a `c` konvertált értéke.

Portabilitás: UNIX kompatibilis.

---

tolower **ctype.h**

Kisbetűre alakít.

```
#include <ctype.h>
int tolower(int c);
```

Megjegyzés: Olyan makróként van megvalósítva, amely az argumentumát kétszer értékeli ki.

Visszatérési érték: `c` konvertált értéke.

Portabilitás: ANSI C és UNIX kompatibilis.

---

toupper **ctype.h**

Nagybetűre alakít.

```
#include <ctype.h>
int toupper(int c);
```

Megjegyzés: Olyan makróként van megvalósítva, amely az argumentumát kétszer értékeli ki.

Visszatérési érték: `c` konvertált értéke.

Portabilitás: ANSI C és UNIX kompatibilis.

`ungetc` **`stdio.h`**

Egy karaktert ír vissza az input folyamba.

```
#include <stdio.h>
int ungetc(int c, FILE *stream);
```

Visszatérési érték: A `c` karakter, hiba esetén `EOF`.

Portabilitás: ANSI C és UNIX kompatibilis.

`write` **`io.h`**

Alacsony szintű file-írás.

```
#include <io.h>
int write(int handle, void *buf, unsigned len);
```

Megjegyzés: `len` byte-ot ír a `handle`-vel azonosított file-ba `buf`-ból.

Visszatérési érték: A sikeresen kiírt byte-ok száma, hiba esetén -1.

Portabilitás: ANSI C és UNIX kompatibilis.

Irodalomjegyzék

[1] B. W. Kernighan – D. M. Ritchie. *A C programozási nyelv.* Műszaki Könyvkiadó., 1985. Fordította Dr. Siegler András.

[2] B. W. Kernighan – D. M. Ritchie. *The C Programming Language.* Prentice Hall, 1988. Second Edition.

[3] B. Stroutstrup. *The C++ programming language.* Addison-Wesley, 1986.

[4] M. A. Ellis – B. Stroutstrup. *The Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[5] Microsoft Windows Software Development Kit. *Guide to Programming.* Microsoft Corp., 1990. Version 3.0.

[6] Microsoft Windows Software Development Kit. *Reference Volume 1–2.* Microsoft Corp., 1990. Version 3.0.

[7] IBM Systems Application Architecture. *Common User Access Advanced Interface Design Guide.* IBM Corp., 1989. Document Number: SY0328-300-R00-1089.

[8] BORLAND C++. *Getting Started.* Borland International, Inc., 1991. Version 2.0.

[9] BORLAND C++. *User's Guide.* Borland International, Inc., 1991. Version 2.0.

[10] BORLAND C++. *Programmer's Guide.* Borland International, Inc., 1991. Version 2.0.

[11] BORLAND C++. *Library Reference.* Borland International, Inc., 1991. Version 2.0.

Keresse könyveinket!

COMPUTERBOOKS

1126 BUDAPEST, TARTSAY V. u. 12.

Tel.: 17-51-564, 17-53-591