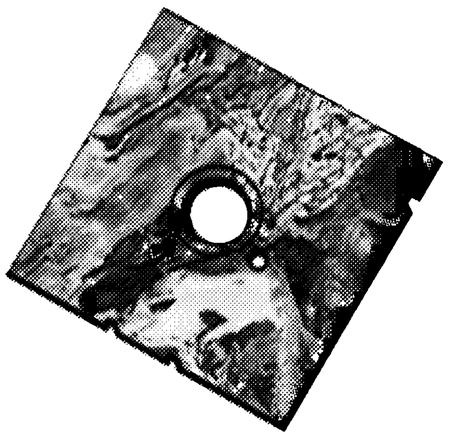
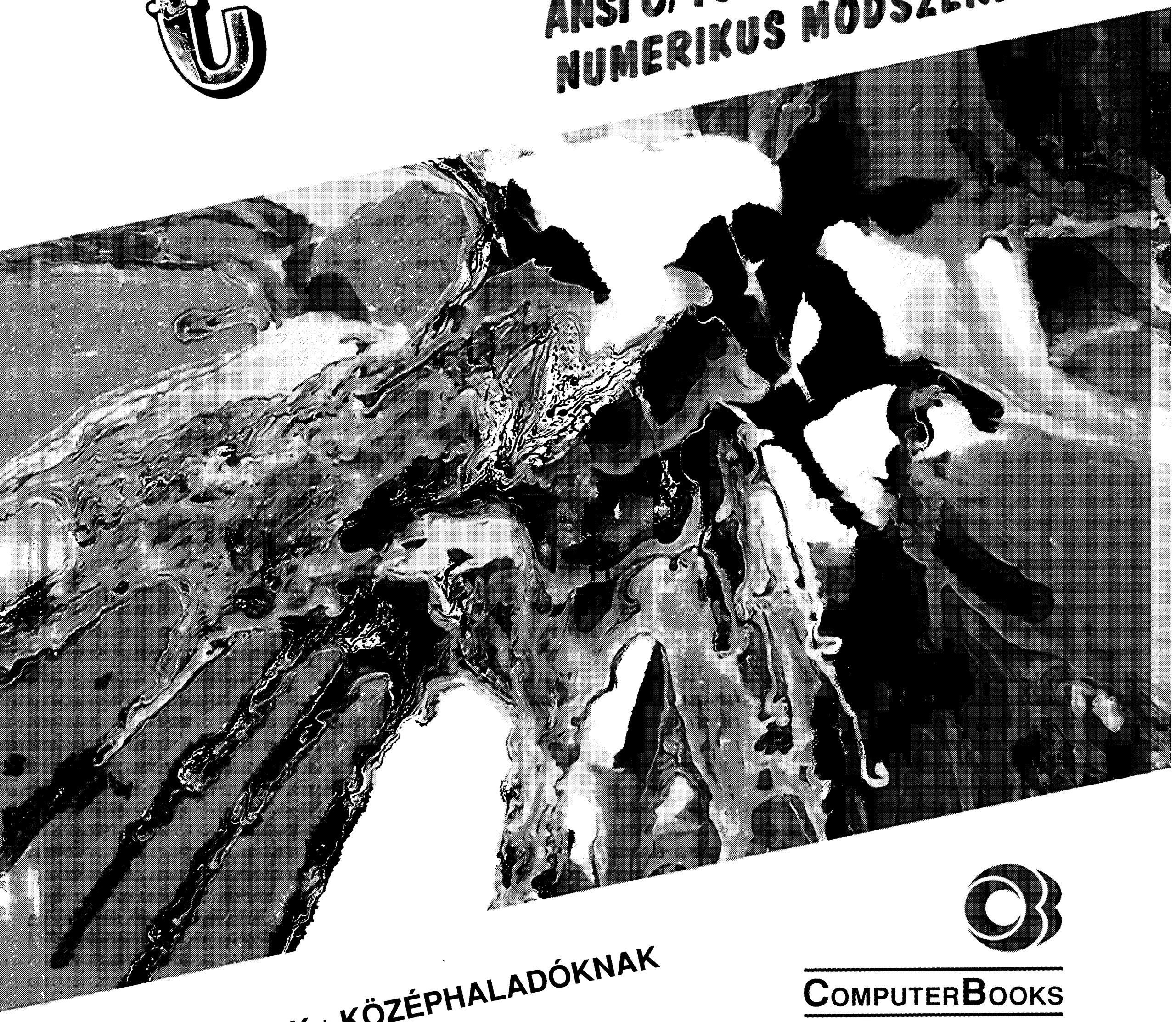


PROGRAMOZZUNK C NYELVEN!

ANSI C, TURBO C, GRAFIKA, NUMERIKUS MÓDSZEREK



KEZDŐKNEK * KÖZÉPHALADÓKNAK

ComputerBooks

PROGRAMOZZUNK C NYELVEN!

A szoftverek illegális használatát a törvény szigorúan bünteti

A számítógépes programok - szövegszerkesztők, táblázatkezelők, adatbáziskezelők, illusztrációs, vagy CAD szoftverek a jelen és a jövő század eszközei, amelyek az Ön munkáját is hatékonyabbá, vállalkozását versenyképesebbé teszik

Magyarországon a nagyvállalatoknál és a kisebb cégeknél használt minden száz szoftver közül 87 illegálisan terjesztett példány 1993 óta a Büntető törvénykönyv 329/A § értelmében a szoftverek illegális használata akár 5 évig terjedő börtönbüntetéssel is sújtható

Ha kétségei vannak az Ön által használt szoftverek jogtisztaságával kapcsolatban, akkor hívja fel a 322 48 91 telefonszámon a BSA Magyarország forródrót szolgálatát, és kérje az ingyenes "Szoftvergazdálkodási Útmutató" című kiadványunkat

Ne másolja, vásárolja!



BSA Business Software Alliance Magyarország

BENKŐ TIBORNÉ
BENKŐ LÁSZLÓ
TÓTH BERTALAN

PROGRAMOZZUNK C NYELVEN!

ANSI C, TURBO C, GRAFIKA, NUMERIKUS MÓDSZEREK

KEZDŐKNEK * KÖZÉPHALADÓKNAK

LEKTOR
HORVÁTH SÁNDOR

COMPUTER BOOKS
BUDAPEST, 1995

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

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

© Tóth Bertalan, Benkő László, Benkő Tiborné, 1994,1995

© Kiadó: ComputerBooks Kiadói Kft
1126 Bp , Tartsay Vilmos u 12
Tel : 175–15–64, tel /fax: 175–35–91
Felelős kiadó: ComputerBooks Kft ügyvezetője
ISBN : 963 618 051 2
Borítóterv: Székely Edith

Köszönetnyilvánítás

Ezúton szeretnénk köszönetet mondani ***Dr. Németh Pálnak***, a Budapesti Műszaki Egyetem Elektronikai Technológiai Tanszék adjunktusának a numerikus módszerek oktatóanyagának összeállításáért.

Dr. Németh Géza, a matematikai tudományok kandidátusa, a KFKI tudományos főmunkatársa szintén hozzájárult ahhoz, hogy a numerikus módszerek fejezet minél érthetőbb legyen, értékes tanácsaiért mondunk köszönetet.

Eppel Gábor a numerikus fejezet példaprogramjai tesztelésében volt segítségünkre.

Kuzmina Jekatyerinának, a BME Informatikai Laboratórium aspiránsának a *"C nyelv lépésről-lépésre"* című fejezet kidolgozásához nyújtott hathatós segítségéért és a könyv ábraanyagának megrajzolásáért mondunk köszönetet.

Végül, de nem utolsósorban ***Horváth Sándor*** lektornak köszönjük az értékes észrevételeit.

Tartalomjegyzék

Előszó **1**

1. Bevezetés **3**

1.1 A C nyelv múltja, jelene és jövője 3

1.2 Gondolatok a C nyelvről 5

2. Ismerkedés a C nyelvvel **9**

3. A C nyelv lépésről-lépésre **23**

3.1. A C nyelv alapelemei **23**

3.1 1. A nyelv jelkészlete 23

3 1.2. A C nyelv azonosítói 24

3.1.3. Konstansok 26

3 1.3.1. Egész konstansok 26

3.1.3.2. Karakter konstansok 27

3.1.3.3. Lebegőpontos konstansok 28

3.1 4. Sztring literálok 29

3 1.5. Megjegyzések 30

3 1.6. Operátorok 31

3.1.7 Írásjelek 31

3.2. A C program szerkezete **33**

3.2 1. A legegyszerűbb C program 33

3.2.2 Egy szöveget kiíró C program 33

3.2.3 Egyetlen modulból felépülő C program 34

3.2.4. Több modulból álló C program 35

3.3. Típusok, változók, konstansok **38**

3.3 1. A C nyelv típusai 38

3.3.1.1. Típuselőírások, típusmódosítók 40

3.3.1.2. Típusminősítők 41

3.3.1.3. A karakter típus 42

3.3.1.4. Egész típusok 42

3.3.1.5. A felsorolt típus 43

3.3.1.6. A lebegőpontos típusok 44

3.3.2. Egyszerű változók definiálása 45

3.3.3 Saját típusok előállítása 47

3.3.4. Konstansok a C nyelvben 47

3.3.5. Értékek, címek és mutatók 49

3.3.5.1. Balérték és jobbérték 49
3.3.5.2. Ismerkedés a mutatókkal 50
3.3.5.3. Mutatók és a dinamikus memóriahasználat 53
3.3.5 4. A void * típusú mutatók 54
3.3.5.5 Többszörös indirektségű mutatók 55

3.4. Operátorok és kifejezések 58
3 4.1. Precedencia és asszociativitás 61
3 4.2. Mellékhatások és a rövidzár kiértékelés 63
3.4.3. Elsődleges operátorok 65
3.4.4. Aritmetikai operátorok 65
3.4.5. Összehasonlító és logikai operátorok 67
3.4.6. Léptető operátorok 69
3.4.7. Bitműveletek 72
3.4.7.1. Bitenkénti logikai műveletek 72
3 4.7.2. Biteltoló műveletek 76
3.4.8. Értékadó operátorok 78
3 4 9. Pointer operátorok 80
3.4.10. A sizeof operátor 82
3 4.11. A vessző operátor 82
3.4.12. A feltételes operátor 83
3.4.13. Típuskonverziók 85
3 4.13.1. Implicit típuskonverziók 85
3.4.13.2. Az explicit típuskonverzió 88

3.5. A C nyelv utasításai 93
3.5 1 Utasítások és blokkok 94
3.5.2. Az if utasítás 95
3 5.2.1. Az if-else szerkezet 97
3 5.2.2. Az else-if szerkezet 99
3 5 3. A switch utasítás 101
3.5.4. A ciklus utasítások 104
3.5.4 1. A while ciklus 105
3.5.4.2. A for ciklus 108
3.5.4.3. A do-while ciklus 111
3.5.5. A break és a continue utasítások 113
3.5.5.1. A break utasítás 114
3.5.5.2. A continue utasítás 115
3.5.6. A goto utasítás 117
3 5.7. A return utasítás 118

3.6. Tömbök, sztringek és mutatók 120
3.6.1. Egydimenziós tömbök 120
3.6.1.1. Egydimenziós tömbök inicializálása 123
3.6.1.2. Egydimenziós tömbök és a typedef 124
3.6.2. Mutatók és a tömbök 125
3.6.3. Sztringek 127
3.6.4. Többdimenziós tömbök 131
3.6.5. Mutatótömbök, sztringtömbök 133
3 6 6. Dinamikus helyfoglalású tömbök 136
3.6.6.1. Dinamikusan létrehozott vektorok 137
3.6 6 2. Kétdimenziós tömbök dinamikus kezelése 139

3.7. Felhasználó által definiált adattípusok 147
3.7.1. A struktúra típus megadása 147
3.7.2. Hivatkozás a struktúra adattagjára 150
3 7.3. Kezdőértékadás a struktúrának 154
3.7 4. Egymásba ágyazott struktúrák 154
3.7.5. Struktúratömbök 156
3.7.6 Union típusú adatstruktúrák 159
3.7.7. A bitmezők használata 163
3 7 8. Önhivatkozó struktúrák használata
- a listaszerkezet 166

3.8. Függvények 173
3.8.1. Függvények definíciója 174
3.8.1.1 A függvények tárolási osztálya 175
3.8.1.2. A függvények visszatérési típusa 176
3.8.1.3. Függvényattribútumok használata 177
3.8.1.4. A függvények paraméterei 178
3.8.2. Függvények deklarációja és prototípusa 179
3.8.3. A függvényhívás 181
3.8.4. A függvény mint függvény argumentum 186
3.8.4.1. A függvénytípus és a typedef 186
3.8.4 2. Függvényre mutató pointerek 187
3.8.4 3 Függvényre mutató pointerek tömbje 190
3 8 5. Struktúra átadása függvénynek 195
3.8.6. Tömb argumentumok használata 197
3.8.6.1. Vektor argumentumok 197
3.8.6.2. Kétdimenziós tömb argumentumok 200
3.8.6.3. Sztring argumentumok 201
3.8.6.4. Konstans paraméterek 205
3.8 7. A main függvény paraméterei és visszatérési értéke 207
3.8.8. Rekurzív függvények használata 210
3.8.8.1. A rekurzív alprogramok csoportosítása 213

3.8 9. Változó hosszúságú argumentumlista 216
3.8.9.1. MS-DOS specifikus feldolgozás 217
3.8.9.2. Szabványos feldolgozás 219
3 8.9.3. Az argumentumlista továbbadása 220
3.8.10. C deklarációk értelmezése és készítése 221
3.8 10.1. C deklarációk értelmezése 222
3.8.10.2. C deklarációk készítése 225

3.9. Tárolási osztályok 228
3.9.1. Az azonosítók élettartama 228
3.9.2. Érvényességi tartomány és a láthatóság 230
3.9.3. A kapcsolódás 232
3.9.4. A névterületek 233
3 9 5. A tárolási osztályok használata 234
3.9.5.1. Az auto tárolási osztály 236
3.9.5.2. Az extern tárolási osztály 239
3.9.5.3. A static tárolási osztály 243
3.9.5.4. A register tárolási osztály 246

3.10. Az előfeldolgozó 248
3.10.1. A C program fordításának fázisai 249
3.10 2 File-ok beépítése a forrásprogramba 250
3.10 3. Makrok használata 250
3 10.3.1 Objektumszerű makrok készítése 251
3.10.3.2. Függvényszerű makrok készítése 252
3 10 3.3. Előredefiniált makrok 257
3 10.4 Feltételes fordítás 258
3 10.5. A #line, az #error és a #pragma direktívák 262

4. Programozás Turbo C könyvtári függvények felhasználásával 265

4.1. Alapvető be- és kiviteli függvények 265
4.1.1. A getchar és a putchar makrok 268
4 1.2. A gets és puts függvények 269
4.1 3. Formázott adat be- és kivitel 269
4.1.3.1. A printf függvény 270
4.1.3.2. A scanf függvény 274
4 1 4. Írás sztringbe és olvasás sztringből 278
4.1.5. Az stdio és stdout átirányítása 278

4.2. A szabványos file-kezelés alapjai 281
4.2.1. A file-mutató deklarálása 282
4.2.2. A file megnyitása 282
4.2.3. A file lezárása 284

4.2.4. Adatátviteli pufferek kijelölése 284
4.2.5. Szöveges file kezelése 286
4.2.6. Bináris file kezelése 287
4 2.7. Pozicionálás a file-ban 289
4 2.8. Hibakezelés 290

4.3. Karakterek osztályozása - adatkonverziós 292
4.3 1. Karakterek osztályozása 292
4.3.2. Karaktert átalakító függvények és makrok 294
4.3.4. Konverziós függvények 294
4.3.3.1. Sztring átalakítása numerikus értékké 294
4.3.3.2. Numerikus értékek sztringgé átalakítása 296

4.4. Puffer és sztringkezelő függvények 298
4.4.1. Pufferek kezelése 298
4 4.2. Sztringek kezelése 299

4.5. Matematikai függvények 302
4.5 1. Trigonometrikus függvények 302
4.5.2. Hiperbolikus függvények 303
4.5.3 Exponenciális és logaritmikus függvények 304
4.5.4 Különféle egyéb függvények 304
4.5.4.1. Abszolút érték függvények 304
4.5.4.2. Hányados és maradék függvények 305

4.6. Memóriakezelés Turbo C rendszerben 308
4.6.1. Memóriamodellek 308
4.6.2. A dinamikus memóriakezelés függvényei 313

4.7. Speciális könyvtári függvények 316
4.7 1. Rendezés és keresés 316
4.7.2. Időfüggvények kezelése 317

4.8. A szöveges képernyő kezelése Turbo C függvényekkel 319
4.8 1. Képernyővezérlő típusok 319
4.8.2. A szöveges mód képernyőablaka 320
4.8.3. Programozás szöveges módban 322
4.8.3.1. Szöveg kiírása és kezelése 322
4.8.3.2. Ablak létrehozás és üzemmód beállítás 324
4.8.3.3. Tulajdonságok beállítása 325
4.8.3.4 Működési információk lekérdezése 326
4.8.3.5. Hangkeltés és program futásának felfüggesztése 327
4.8.4. A szöveges mód konstansai 327
4.8.5 Mintaprogramok a szöveges mód használatára 329
4.8.5.1. Szöveges ablakok használata 329
4.8.5.2. Adat beolvasása és ellenőrzése 332

4.9. Grafikus képernyő kezelése Turbo C függvényekkel.....338
4 9.1. A grafikus koordinátarendszer.....338
4.9.2. Az aktuális pointer (grafikus kurzor).....339
4 9 3. Kiírások a grafikus képernyőn.....340
4.9.4. Képernyőlapok és színek.....340
4.9.5. Hibakezelés.....340
4.9.6. A grafikus könyvtár függvényeinek használata.....340
4.9.6.1. A grafikus üzemmód aktivizálása.....340
4.9.6.2. Visszatérés a szöveges üzemmódba.....344
4.9.6.3. Színek használata.....344
4 9.6.4. Rajzolási módok.....346
4.9.6.5. Viewport beállítások.....346
4.9.6.6. Rajzolási módok.....347
4.9.6.7. Vonaltípusok.....348
4.9.6 8. X:Y viszony.....349
4.9 6.9. Festési módok.....349
4 9.7. Rajzolás a grafikus képernyőre.....350
4.9.7 1 A grafikus kurzor (aktuális pozíció, tollhegy).....351
4.9.7 2. Pont rajzolás.....351
4.9.7.3. Egyenes vonal rajzolása.....351
4.9 7.4. Görbe vonalak.....352
4.9.7.5. Festés.....353
4.9.7.6. Törlés a képernyőn.....355
4.9.7 7. Bitminták a képernyőn.....355
4.9.8. Szövegek a grafikus képernyőn.....356
4.9.9. Hibakezelés.....358
4.9.10. A grafikus rendszer további lehetőségei.....359
4 9.10.1. Az initgraph függvény működésének módosítása.....359
4.9.10.2. A grafikus vezérlők és a karakterkészlet beszerkesztése a futtatható programba.....359
4.9 10.3. Grafikus rendszer bővítése új vezérlőkkel és karakterkészlettel.....360
4.9 11. A grafikus könyvtár függvényeinek csoportosítása.....361
4.9.11.1. Grafikus rendszer vezérlése.....361
4.9.11.2. Rajzolás és festés.....361
4.9.11.3. Képernyő, viewport, image és pixel.....362
4.9.11.4. Szöveg kiírása grafikus módban.....363
4.9.11.5. Színvezérlés.....363
4.9.11.6. Hibakezelés grafikus módban.....364
4.9.11.7. Állapot lekérdezése.....364
4.9.12. Grafikus programok készítése.....365
4.9.12.1. Téglalap rajzolása.....365

4 9.12.2. Szöveg kiírása grafikus módban.....367
4 9.12.3. A szöveg pozicionálása.....371
4 9.12.4. A szöveg szélességének változtatása.....371
4.9.12.5. A meghajtó nevének kiíratása.....372
4 9.12.6. A szöveges és a grafikus mód váltása.....372
4.9.12.7. Alakzat mozgatása.....373
4.9.12.8. Képernyő torzításának kiküszöbölése.....375
4.9.12.9. Alakzatok rajzolása.....376
4.9.12.10. Kép kivágása és áthelyezése.....377
4.9.12.11. Ferdehajítás grafikus ábrázolása.....379
4 9.12.12. Grafikus kurzor mozgatása.....380

5. Numerikus módszerek és a C nyelv.....383
5.1 Lineáris egyenletrendszer megoldása.....385
5.1.1. Gauss-féle kiküszöbölési eljárás.....386
5.1.2. Gauss-Jordan módszer.....395
5.1.3. Fokozatos közelítések módszere.....396
5.1.4. Seidel módszer.....397
5.1.5. Lineáris egyenletrendszer megoldása LU dekompozícióval.....398
5.1.6. Mátrixinvertálás.....406
5.2. Egyismeretlenes nemlineáris egyenlet megoldása.....410
5.2.1. Gyök behatárolása intervallum-felezéssel.....412
5.2.2. Gyök meghatározása érintő módszerrel (Newton-Raphson módszer).....414
5.2.3. Húr módszer.....416
5.2.4. Gyök meghatározása a Newton-Raphson módszer és a húr módszer együttes alkalmazásával.....418
5.2.5. Gyök meghatározása szelő módszerrel.....420
5.2.6. Gyök meghatározása fokozatos közelítéssel.....421
5.3. Interpoláció, regresszió.....424
5.3.1. Interpoláció.....425
5.3.2. Lineáris interpoláció.....426
5.3.3 Lagrange interpoláció.....427
5.3.3.1. Elsőfokú Lagrange féle interpolációs polinom.....429
5.3.4 Aitken interpoláció.....431
5.3.25 Regresszió.....433
5.4. Numerikus integrálás (numerikus kvadratúra).....438
5.4.1. Newton-Cotes módszerek.....440

5.4.1.1. Téglalap formula 440
5.4.1.2. Egyszerű (kis) trapéz formula 442
5.4.1.3. Összetett (nagy) trapéz formula. 443
5.4.1.4. Érintő formula 445
5 4.1.5. Egyszerű (kis) Simpson formula 446
5.4.1 6. Összetett (nagy) Simpson formula 447

5.4.2. Romberg eljárás 449
5.4.3. Nem ekvidisztáns osztású kvadratúra:
Gauss és Csebisev formulák. 450

F1. A Turbo Pascal és a Turbo C nyelv összehasonlítása 453
F1.1. A program szerkezete 453
F1.2. A programozás elemeinek összehasonlítása 456
F1 2.1. Az output művelet 456
F1.2.2. Adattípusok 458
F1.2.3. Műveletek 459
F1.2.4. Adatbevitel 460
F1.2.5. A blokk utasítás 460
F1.2.6. Feltételes utasítások 461
F1.2.7. Ciklusutasítások 463
F1.2.7 1. A while ciklus 464
F1.2.7.2. A do..while ciklus 464
F1.2.7.3. A for ciklus 465
F1.2.8. Alprogramok 466
F1.2.9. C függvények prototípusa 468
F1.3. Az adatstruktúrák áttekintése 469
F1 3.1. Mutatók 469
F1.3.2. Tömbök 471
F1.3.3. Sztringek 472
F1.3.4. Struktúrák 474
F1.3.5. Unionok 475
F1.4. Programozásbeli különbségek 476
F1.4.1. Betű-érzékenység 476
F1.4 2. Típuskonverziók (type-casting) 476
F1.4.3. Konstansok, változók tárolása, kezdőérték-adása. 477
F1.4.3.1. Konstans típusok 477
F1.4.3.2. Változó kezdőértéke 477
F1.4.3.3. Változók tárolása 478
F1.4.3.4. Dinamikus memóriafoglalás 478
F1.4.4. Parancssor argumentumok 479
F1.4.5 File I/O 480

F2. Turbo C 2.0 include file-ok és könyvtári függvények 483
F2 1 A Turbo C 2 0 include file-jai 483
F2 2 Globális változók és szabványos típusok 485
F2.2 1 Globális változók 485
F2 2 2 Szabványos típusok 487
F2 2 3 A BGI könyvtár globális változói és típusai 489
F2 3 A könyvtári függvények csoportosítása 491
F2 3 1 Karakterek osztályozása és konverziója 491
F2 3 2. Adatkonverziók 492
F2 3 3 Könyvtárak kezelése 492
F2 3.4. Input és output rutinok 493
F2.3.4.1. Stream rutinok 494
F2 3 4.2 Alacsonyszintű I/O rutinok 496
F2.3 4.3. Konzol és port I/O 498
F2.3 5 Matematikai rutinok 499
F2.3 6 Memóriakezelés 501
F2 3 7. Pufferek használata. 502
F2.3 8 Sztringkezelés 502
F2.3 9. Folyamatvezérlés 504
F2 3 10. Időhasználat 506
F2.3 11 Változó hosszúságú argumentumlista kezelése (makrok) 507
F2.3.12. Keresés és rendezés 507
F2.3 13 További szabványos függvények. 507
F2 3.14. Rendszerhívások 508
F2.3.14.1 BIOS interface 508
F2 3.14 2 MS-DOS interface 509
F2 3 15. A BGI grafikus könyvtárhoz tartozó függvények 511
F2 4 A Turbo C 2.0 nyelv kulcsszavai 515

F3. Függvények grafikus módban 517

F4. A Turbo C fejlesztői rendszer kezelése 539
F4 1 Szövegszerkesztés 539
F4.2. Fordítás, szerkesztés és futtatás 539
F4 3. Project fogalma és használata 540

F5. A lemezmelléklet használata 543

F6. Összefoglaló táblázatok 545

Irodalomjegyzék 549

Tárgymutató 551

Előszó

Kinek szánjuk a könyvet?

A "*Programozzunk C nyelven!*" című könyv elsősorban azok számára íródott, akik most kezdenek ismerkedni a programozási nyelvek "angoljának" nevezhető C nyelvvel. A könyv felépítése olyan, hogy nem szükséges a teljes művet elolvasni ahhoz, hogy a feldolgozott témakört programok írásával mélyítse el az Olvasó. Amennyiben a feldolgozást sikeresnek ítéli meg, tovább lehet lépni a következő fejezet olvasásával. Az önellenőrzéshez a fejezetek végén található kérdések és feladatok nyújtanak segítséget.

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

A könyvhöz csatolt lemezmelléklet nemcsak megkíméli az Olvasót a példák időt rabló begépelésétől, hanem a kitűzött feladatok megoldását is tartalmazza. A lemezmelléklet *CMIX* állománya a legkülönbözőbb alkalmazási területekről hoz példákat, melyekre a könyvünkben a korlátozott terjedelem miatt nem térhettünk ki.

Miről szól a könyv?

A könyv felépítése olyan, hogy folytonosan olvasva, meg lehet ismerkedni a C nyelvvel és a legfontosabb könyvtári függvényekkel. A "*Bevezetés*" fejezetben bemutatjuk a C nyelv múltját és jelenét. A következő fejezetben az olvasóval együtt oldunk meg feladatokat C nyelven, ízelítőt adva a C nyelv lehetőségeiből.

A harmadik fejezet az ANSI C nyelv tankönyvének tekinthető, hisz az alfejezetek olvasásával lépésről-lépésre juthat előre az Olvasó a C nyelv csodálatos világában. A következő nagy fejezet tematikus csoportosításban bemutatja a Turbo C 2.0 rendszer könyvtári függvényeinek használatát,

kezdve az alap adat be- és kiviteltől, a memóriakezelésen keresztül a grafikus lehetőségek bemutatásáig.

Külön érdekessége a könyvünknek az a néhány numerikus módszer, amelyeknek mind a matematikai alapjait, mind pedig a programozását tartalmazza az ötödik fejezet. Ez segítséget nyújthat az ilyen jellegű feladatok megoldásához. Továbbá kiváló programozási gyakorlat az is, ha az Olvasó saját programváltozatot dolgoz ki a leírt módszerek megoldására.

A könyvet gazdag függelék zárja. Nagy segítséget nyújthat a Pascal nyelvet használó programozók számára a C nyelv elsajátításában az F1. függelék, amely a Turbo Pascal és a Turbo C nyelvek összehasonlítását tartalmazza. A teljes Turbo C 2.0 könyvtárat csoportosító F2. függelékben információk találhatók arról, hogy a függvények megtalálhatók-e az ANSI C és a UNIX C nyelvekben. Ez hasznos segítség azoknak, akik komolyan gondolják a hordozható C programok fejlesztését.

Mitől más ez a könyv?

A C nyelv hazai (magyar nyelvű) irodalma kielégítőnek nevezhető, azonban ez a könyv az első olyan magyar nyelvű mű, amely a C nyelv szabványosított változatát, az ANSI C nyelvet ismerteti.

A könyv megírását évtizedes oktatási gyakorlat előzte meg a Budapesti Műszaki Egyetem Villamosmérnöki Karán és Mérnöktovábbképző Intézetében, amely során szerzett tapasztalatok nagyban hozzájárultak könyvünk szerkezetének és tartalmának kialakításához Ezúton is köszönetet szeretnénk mondani annak a több száz hallgatónak, akik közvetve ugyan, de hozzájárultak könyvünk megjelenéséhez.

A szerzők

1. Bevezetés

A C általános célú programozási nyelv. A C nyelv szorosan kapcsolódik a UNIX operációs rendszerhez, azonban valójában egyetlen operációs rendszer vagy számítógéptípus sem sajátíthatja ki azt. A C nyelvet, mint "rendszerprogramozási nyelvet" szokták emlegetni, mivel jól használható operációs rendszerek és fordítóprogramok írására, ugyanakkor kiválóan alkalmas tetszőleges programozási feladat megoldására.

1.1 A C nyelv múltja, jelene és jövője

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

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

Több éven át csak az **AT&T UNIX** operációs rendszerrel szállított C-fordító képviselte a C nyelv definícióját. Az 1978-ban megjelent ***B.W. Kernigham*** és ***D. M. Ritchie*** által írt *"The C Programming language"* című könyv szolgált a nem **UNIX**-alapú **C**-implementációk alapjául. (Ezt a művet, illetve ennek a második kiadását *K&R*-el jelöli a szakirodalom, mely magyar fordításban csak 1985-ben látott először napvilágot, a *Műszaki Könyvkiadó* gondozásában.)

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

Az amerikai szabványügyi hivatal (***ANSI***) 1983-ban létrehozta az ***X3J11*** bizottságot a C nyelv szabványos változatának kidolgozására. A bizottság tagjai különböző számítógépes cégek képviselői voltak. A **ANSI C** szabvány végleges változata (*ANSI X3 159-1989*) hat évi munka után 1989-ben készült el. A szabványosítás során az alábbi elveket tartották szem előtt a bizottság tagjai:

- A létező C nyelvű programok lefordíthatók maradjanak a szabványos C nyelvet megvalósító fordítóval. A nem szabványos megoldásokra figyelmeztetés hívja fel a programozó figyelmét.
- A C forráskód legyen hordozható.
- A C forráskód lehet nem hordozható is. (A szabványos C bővíthető géptől és operációs rendszertől függő megoldásokkal.)
- Tartsa meg a C szellemiségét, amit a következő kifejezések híven tükröznek:
  - "Bízz a programozóban!"
  - "Ne akadályozd a programozót a munkájában, hiszen ő tudja mi az amit tennie kell!"
  - "A nyelv legyen kicsi és egyszerű!"
  - "Valamely művelet elvégzésére csak egy utat biztosíts!"
  - "Tedd gyorssá a programot, még akkor is, ha ez nem biztosítja a kód hordozhatóságát!"

A C nyelv történetét itt le is zárhatnánk, azonban tudnunk kell azt, hogy napjainkban újabb C-fordítók már nem születnek. A C nyelv helyét, szerepét világszerte a **C++** nyelv veszi át.

A **C++** nyelv fejlesztése szintén az ***AT&T Bell Laboratóriumok*** cégnél kezdődött a 80-as évek elején és még napjainkban is tart. A C++ nyelv kidolgozásánál, amely ***Bjarne Stroustrup*** nevéhez fűződik, alapvető cél volt a C nyelv kiegészítése a **SIMULA 67** nyelvből átvett osztályfogalommal. Ez lehetővé teszi az ***objektum-orientált*** programozási módszer használatát. Összességében elmondható, hogy a C++ önálló, a C nyelv jelölésrendszerére épülő programozási nyelv, amely sok hasznos megoldással bővíti a C nyelvet.

A C programok többsége minden további nélkül átvihető C++ rendszerbe. Mivel a C++ nyelv *"erősen típusos nyelv"*, a szükséges módosításokat általában a konverziók és a deklarációk területén kell elvégezni. A legtöbb C++ fordító azonban képes C fordítóként is működni.

Ugyancsak elmondható, hogy a C nyelv természetes kiinduló pont a C++ nyelv megismeréséhez. Jelen könyvünkben csak a C nyelv ismertetésére szorítkozunk, hiszen a C++ nyelv teljes leírása egy további könyvet igényel.

1.2 Gondolatok a C nyelvről

A C nyelv történetének áttekintése után dióhéjban összefoglaljuk a nyelv jellegzetességeit, melynek áttanulmányozásával más programozási nyelveket ismerő Olvasó összefoglaló képet kaphat a C-ről.

Míg a BCPL és a B nyelvek típus nélküli nyelvek voltak, addig a C nyelv több adattípussal is rendelkezik. A alap adattípusok a karakteres, az egész és a lebegőpontos típusok többféle méretben is elérhetők. Az alaptípusok mellett ún. származtatott (*derived*) típusokat is használhatunk, mint a mutatók, a tömbök, a struktúrák és az uniók. A különböző típusok között automatikus konverziók definiáltak, ezért a C nyelvet szokás *"nem típusos"* jelzővel ellátni. A nyelv kifejezései, amelyek operátorokkal összekapcsolt operandusokból állnak (de ide tartozik az értékadás és a függvényhívás is), pontosvesszővel lezárva utasításokat definiálnak. A mutatók segítségével gépfüggetlen címaritmetikát valósít meg a C nyelv.

A C nyelv alapvető programvezérlő konstrukciókat is tartalmaz, amelyek felhasználásával jól strukturált program készíthető: utasításcsoportok kialakítása ({ }), kétirányú elágaztatás (**if-else**), valamely eset kiválasztása az esetek lehetséges halmazából (**switch**), tevékenység ismétlése belépési feltétel

(**while**, **for**), illetve kilépési feltétel (**do-while**) megadásával. A ciklusok működése a **break** (kiugrás) és a **continue** (következő iteráció) utasítások segítségével finoman szabályozható.

A C nyelv blokk-strukturált ugyan, azonban a függvényeket (alprogramokat) nem lehet egymásba építeni (mint pl. a Pascal nyelvben). A függvények tetszőleges alap-, struktúra, unió vagy mutató típusú függvényértékkel rendelkezhetnek. A nyelv támogatja a függvények rekurzív hívását.

A C nyelven történő programfejlesztéskor a modularitás elvét ajánlott követni. A tárolási osztályok definiálják az egyes modulok, illetve adott modulon belül a függvények között a változók és a függvények elérhetőségét és élettartamát.

A megfelelő forrásprogram felépítését az ún. előfordító (*preprocessor*) segíti, a szöveghelyettesítés (*makro*), a szövegbeépítés (*include*) és a feltételes fordítás mechanizmusával.

A C nyelvet szokás "*alacsony szintű*" nyelvként is emlegetni, ami azonban nem pejoratív értelemben értendő. Egyszerűen arról van szó, hogy a C nyelv a legtöbb számítógép által használt objektumokkal (karakterek, számok és címek) dolgozik. Ezen objektumok között olyan műveletek (bitműveletek, címaritmetika) elvégzése is lehetséges, amelyeket más *magas szintű* programozási nyelvek nem támogatnak. Ugyancsak a C nyelv sajátossága a bit-struktúrák kezelése.

A C nyelv definíciója nem tartalmaz közvetlen műveleteket karaktersorozatok (sztringek), halmazok, listák és tömbök kezelésére. (Azonban a struktúrák értékadással történő másolása megoldott.)

A C az adat be- és kivitel (*I/O*) tekintetében szintén nem tartalmaz utasításokat (mint például a FORTRAN vagy a Pascal). A fenti műveletek elvégzésére eljárásgyűjtemény (függvénykönyvtár) áll a C programozó rendelkezésére. Ez a megoldás sokkal rugalmasabb a hagyományos nyelvek (FORTRAN, BASIC) utasításainál, sőt a később kifejlesztett nyelvek (mint pl. a Modula-2) szintén ezt az utat követik.

Könyvünkben a C nyelv bemutatása során különválasztjuk a nyelv definíciójának és könyvtári függvényeinek ismertetését. A könyvünk első részében az ANSI C nyelv definíciójára támaszkodva lépésről-lépésre tárjuk fel a C nyelv elemeit, lehetőségeit. Ezt követi a könyvtári függvények tematikus csoportosítása, ahol a Turbo C 2.0 fordító könyvtári függvényeit használjuk a példaprogramokban.



BUDAPESTI MŰSZAKI EGYETEM
Mérnöktovábbképző Intézet

Sokoldalú számítástechnikai képzés egyetemi oktatók és nagy gyakorlattal rendelkező szakemberek részvételével

Saját géptermek. Egy hallgató - egy gép.

Maximum 15 fős gyakorlati csoportok. Mérsékelt árak.

Több, mint 80 tanfolyam az alábbi témakörökben:

Számítógép-kezelői ismeretek
Adatbáziskezelés
Szoftverfejlesztés
MS DOS alkalmazások
Térinformatika
Multimédia
Számítógéphálózatok

Valamint a **Novell Oktatóközpont** kínálata:

Általános ismeretek
NetWare operációs rendszerek
NetWare és UNIX (TCP/IP) kapcsolata
UnixWare ismeretek
Távoli hálózati termékek
Üzemeltetési és szervízismeretek

Részletes felvilágosítás a 204-1111/34-97-es és a 204-1111/29-57-es telefonszámon!

2. Ismerkedés a C nyelvvel

Valamely programozási nyelv elsajátításának legbiztosabb módszere az, ha programot írunk az adott nyelven. Ebben a fejezetben egy egyszerű feladat megoldásán keresztül kívánjuk bemutatni az Olvasónak a C nyelv lehetőségeit. Természetesen csak akkor lesz igazán sikeres az ismerkedés, ha a számítógépén begépeli és lefuttatja az egyes példaprogramokat.

A legegyszerűbb C program mindössze egyetlen függvényt, a ***main*** függvényt tartalmazza. Ahhoz, hogy a programunk futásának látható eredménye legyen, írassuk ki a képernyőre a "*C nyelv*" szöveget. Ehhez persze keresnünk kell egy könyvtári függvényt, amely képes a feladat elvégzésére. A keresés az F2. függelék áttanulmányozását jelenti, ahol a "*Stream rutinok*" között meg is találjuk a ***printf*** függvényt. Az alfejezet címe alatt láthatjuk az STDIO.H állomány nevét, amely a ***printf*** függvény deklarációját (prototípusát) tartalmazza. Hibátlan program írásának előfeltétele, hogy minden felhasznált könyvtári függvény deklarációját beépítsük a programunkba az ***#include*** előfordító utasítással:

```
#include <stdio.h>

main()
{
  printf("C nyelv\n");
}
```

A ***main*** függvény törzsét kapcsos zárójelek fogják közre. A ***printf*** argumentumaként a kiírandó szöveget kettős idézőjelek (") között kell megadni (sztring). A szöveg végén álló '\n' karakter hatására a kiírás után új sorba megy a kurzor.

A ***printf*** függvény nevében az 'f' betű a formátum szóból származik. A kiírandó szövegben helyezzük el az 1994-es évszámot. Ezt megtehetjük a

```
printf("C nyelv - 1994\n");
```

utasítás használatával, de a későbbi teendőinket figyelembe véve, használjuk a tetszőleges egész szám kiírására alkalmas megoldást:

```
printf("C nyelv - %d\n",1994);
```

Meg kell jegyeznünk, hogy a formátumsztringben ("C nyelv - %d\n") minden típushoz külön formátumelem tartozik. A "%d" formátum csak egész (**int**) számok esetén ad helyes eredményt.

A következőkben módosítsuk úgy a programunkat, hogy az alkalmas legyen két tetszőleges egész szám és azok összegének kiírására:

```
#include <stdio.h>

main()
{
 printf("A két szám összege: %d + %d = %d\n", 7, 30, 7+30);
}
```

A program futtatásának eredménye:

```
A két szám összege: 7 + 30 = 37
```

Programot legtöbbször általános céllal készítünk, amihez olyan nevek bevezetése szükséges, amelyek által kijelölt memóriaterület tartalma megváltoztatható. A példánkban használjuk az *a*, *b* és *c* változókat. Minden változó rendelkezik típussal, amely az általa használt memóriaterület méretét definiálja. A változóknak értéket is adhatunk az értékadás (=) operátorának alkalmazásával.

```
#include <stdio.h>

main()
{
  int a,b,c;
  a=7;
  b=30;
  c=a+b;
  printf("A két szám összege: %d + %d = %d\n", a, b, c);
}
```

A forrásprogramban megjegyzéseket is elhelyezhetünk, amellyel a programban elvégzett műveleteket magyarázzuk. Megjegyzések segítségével valamely program működése több hónap távlatából is azonnal értelmezhető.

A C nyelvben a megjegyzéseket a `/*` és a `*/` karakterek zárják közre.


```
#include <stdio.h>

main()
{
  int a,b,c;       /* A változók definiálása */
  a=7;             /* Értékadás */
  b=30;
  c=a+b;           /* Az összeg kiszámítása */
  /* Kiíratás */
  printf("A két szám összege: %d + %d = %d\n", a, b, c);
}
```

A későbbiekben eltekintünk a megjegyzések használatától, hisz a program működését a programok közötti összekötőszövegben magyarázzuk.

A változók használatával lehetővé válik, hogy az *a* és *b* változóknak nem a program fordítása, hanem a futása során adjunk értéket. Ezt a műveletet adatbevitelnek (*input*) nevezzük, melynek elvégzésére a ***scanf*** függvény használható.

```
#include <stdio.h>

main()
{
  int a,b,c;
  printf("Kérek két számot: ");
  scanf("%d,%d",&a,&b);
  c=a+b;
  printf("A két szám összege: %d + %d = %d\n", a, b, c);
}
```

A ***scanf*** függvény első argumentuma egy formátumsztring, amely a fenti példában azt jelöli, hogy két vesszővel elválasztott egészet kívánunk a billentyűzetről beolvasni. A következő két argumentummal mondjuk meg, hogy hova tegye a függvény a beolvasott értékeket. Erre a célra az *a*, illetve a *b* változók címét adjuk meg, mint argumentumot. (A & operátort a változó neve előtt használva, a változó címét kapjuk meg.) Természetesen megváltozik a program futási képernyője is:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
```

Az előző program teljesen hibás eredmény ad, ha elfelejtjük a számokat elválasztó vesszőt begépelni vagy, ha rossz számot, például tizedestörtet adunk meg Ezekben az esetekben maga a ***scanf*** képes figyelmeztetni a hibás

adatmegadásra. A függvény részletes leírásából megtudhatjuk, hogy a függvény neve által visszaadott érték (a függvényérték) azt tartalmazza, hogy hány értéket dolgozott fel a függvény az input sorból. A fenti példában az *a* és *b* változó akkor kap értéket, ha a ***scanf*** függvény által visszaadott érték 2. Az ellenőrzés elvégzéséhez az **if** utasítást használjuk.

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

main()
{
  int a,b,c;
  printf("Kérek két számot: ");
  if (scanf("%d,%d",&a,&b)!=2)
     {
       printf("Hibás adatbevitel!\n");
       exit(-1);
     }
  c=a+b;
  printf("A két szám összege: %d + %d = %d\n", a, b, c);
}
```

Az **if** utasítással azt vizsgáljuk, hogy a ***scanf*** függvény által visszaadott érték 2-től különbözik-e. Ha különbözik, akkor program az **if** utáni blokk feldolgozásával folytatódik, amelyben kiírjuk a hibaüzenetet és kilépünk a programból. (A kiléptető ***exit*** függvény prototípusát a STDLIB.H állomány tartalmazza.) Helyes adatok megadása esetén az **if** utáni blokkot átlépi a program és a már megszokott módon fejezi be a futását.

Az ellenőrzés beépítésével a programunk minden igényt kielégít, hisz az összeget csak helyes adatok megadása után írja ki.

Tegyük fel, hogy több számpáros összegét szeretnénk meghatározni. Ekkor elég kényelmetlen minden alkalommal újraindítani a programot. A problémát egyszerűen megoldhatjuk, az elvégzendő tevékenységek (beolvasás, számolás és kiírás) ciklikus ismétlésével. Most már csak egy leállási feltételt kell találnunk, mint amilyen például a szóköz billentyű lenyomása.

A C programnyelvben ciklusokat használunk bizonyos műveletek ismétlésére. A kérdés már csak az, hogy a ciklusban hol teszteljük a kilépési feltételt. Mivel a programot akkor kívánjuk futtatni, ha legalább egy számpáros

összegezésére van szükség, azt a ciklust választjuk ki, amely a végén ellenőrzi kilépési feltételt - ez a **do-while** ciklus.

A ciklus feltételében a ***getch*** függvényt használjuk karakter beolvasására. Ez a CONIO.H-ban deklarált függvény vár egy billentyű lenyomásáig, majd függvényértékként megadja a beolvasott karakter kódját. (Szóközt nyomva kilépünk a ciklusból, míg bármely más billentyű lenyomásával a ciklusunk újabb iterációval folytatódik.) A szóközt a kódjával (32) adtuk meg.

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

main()
{
  int a,b,c;
  do
  {
    printf("Kérek két számot: ");
    if (scanf("%d,%d",&a,&b)!=2)
    {
       printf("Hibás adatbevitel!\n");
       exit(-1);
    }
    c=a+b;
    printf("A két szám összege: %d + %d = %d\n", a, b, c);
  } while  (getch()!=32);
}
```

A program futását az alábbi képernyőrészlet szemlélteti:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: 13,26
A két szám összege: 13 + 26 = 39
Kérek két számot: 4,26
A két szám összege: 4 + 26 = 30
Kérek két számot: 1 2
Hibás adatbevitel!
```

Tehát a programunk mindaddig fut, amíg helyes adatokat adunk meg és nem nyomjuk meg az összeg kiírása után a szóköz billentyűt. Hibás adat megadása esetén azonban azonnal befejeződik a program futása. Hogyan lehet a hibaüzenet kiírása után folytatni a program futását?

A megoldást az ***exit*** függvény hívása helyett a **continue** utasítás használata adja, amellyel a ciklus futását a következő iterációval folytathatjuk.

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

main()
{
  int a,b,c;
  do
  {
    printf("Kérek két számot: ");
    if (scanf("%d,%d",&a,&b)!=2)
    {
       printf("Hibás adatbevitel!\n");
       continue;
    }
    c=a+b;
    printf("A két szám összege: %d + %d = %d\n", a, b, c);
  } while  (getch()!=32);

}
```

Azonban a programot futtatva meglepő eredményt kapunk:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: 1.375 12.34
Hibás adatbevitel!
Kérek két számot: Hibás adatbevitel!
Kérek két számot: Hibás adatbevitel!
Kérek két számot: Hibás adatbevitel!
```

Amennyiben nem a szóközt nyomjuk le, az utolsó sor ismétlődik több lépésen keresztül. Mi a probléma a megoldásunkkal? A program szintaktikája hibátlan, így a hiba okát a színfalak mögött kell keresnünk, méghozzá a ***scanf*** működését kell nagyító alá vennünk. A ***scanf*** függvény számára az adatokat az operációs rendszer az <*Enter*> billentyű lenyomásakor egy ideiglenes tárolóterületre (pufferba) tölti. A ***scanf*** onnan csak annyi karaktert dolgoz fel amennyit értelmezni képes, a többi pedig bennmarad a pufferben. A következő ***scanf***-hívás nem vár az adatok begépelésére, hanem egyből a pufferben maradt karaktereket próbálja értelmezni, természetesen sikertelenül.

A probléma egyszerűen kiküszöbölhető, ha a hiba fellépése esetén az adatbeviteli (*input*) puffert kiürítjük. Erre a szabványos rutinkönyvtár szintén tartalmaz megoldást az ***fflush*** függvény meghívásával. Ehhez azonban azt is kell tudnunk, hogy a ***scanf*** nem közvetlenül a billentyűzetről olvas adatokat, hanem az ***stdin*** szabványos adatfolyamból (*stream*-ből).

```
#include <stdio.h>
#include <conio.h>

main()
{
  int a,b,c;
  do
  {
    printf("Kérek két számot: ");
    if (scanf("%d,%d",&a,&b)!=2)
    {
      printf("Hibás adatbevitel!\n");
      fflush(stdin);
      continue;
    }
    c=a+b;
    printf("A két szám összege: %d + %d = %d\n", a, b, c);
  } while  (getch()!=32);

}
```

A javított program működése, már minden szempontból megfelel a kívánalmaknak.

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: 1.375 12.34
Hibás adatbevitel!
Kérek két számot: 4,26
A két szám összege: 4 + 26 = 30
Kérek két számot: 1 2
Hibás adatbevitel!
```

Az adatbeviteli puffer működésével kapcsolatos problémák megoldására egy általánosabb módszert is alkalmazhatunk, melynek elve nagyon egyszerű. Minden adatbevitelt először sztringbe végzünk el, majd ebből a sztringből olvassuk ki az adatokat a megadott változókba. A sztring beolvasására a ***gets*** függvényt, míg a sztringből történő adatbevitelre az ***sscanf*** függvényt használjuk

```
#include <stdio.h>
#include <conio.h>

main()
{
  int a,b,c;
  char str[160];              /* Tömb a sztring tárolására*/
  do
  {
    printf("Kérek két számot: ");
    gets(str);               /* Adatbeolvasás a sztringbe */
                             /* Olvasás a sztringbõl      */
    if (sscanf(str,"%d,%d",&a,&b)!=2)
    {
       printf("Hibás adatbevitel!\n");
       continue;
    }
    c=a+b;
    printf("A két szám összege: %d + %d = %d\n", a, b, c);
  } while  (getch()!=32);
}
```

Miért használjuk ezt az utóbbi bonyolultabb megoldást az előző egyszerű megoldás helyett? A választ megtaláljuk, ha átgondoljuk, hogy a beolvasott szöveget, például egyszerűbb ellenőrzés elvégzésére, mi magunk is feldolgozhatjuk (a ***sscanf*** meghívását megelőzően).

A következő programban, amennyiben érkezett adat, azt feldolgozzuk és az ***sscanf*** értékének tesztelésével több információt tartalmazó hibaüzenetet írhatunk ki. A példában lehetséges ***sscanf*** értékeket és ezekből levont következtetéseket táblázatban foglaltuk össze:

0 - hibás az első szám,

1 - hibás a második szám,

2 - nincs hiba.

A három különböző eset feldolgozását összetett **if**-es szerkezet segítségével is elvégezhetnénk, azonban sokkal áttekinthetőbb programhoz jutunk, ha a többirányú elágaztatások megvalósítására szolgáló **switch** utasítást használjuk. A program helyes működéséhez meg kell vizsgálnunk, hogy érkezett-e adat, vagy sem. Amennyiben nincs adat, akkor újabb iterációval folytatódik a programunk futása.


```
#include <stdio.h>
#include <conio.h>
main()
{
  int a,b,c;
  char str[160];
  do
  {
    printf("Kérek két számot: ");
    gets(str);
    if (str[0]==0) continue;              /* Van-e adat */
    switch (sscanf(str,"%d,%d",&a,&b)) /* Elágaztatás */
    {
      case 0:
        printf("Hibás az 1. adat!\n");
        break;
      case 1:
        printf("Hibás a 2. adat!\n");
        break;
      default:
        c=a+b;
        printf("A két szám összege: %d + %d = %d\n",a,b,c);
        break;
    }
  } while (getch()!=32);
}
```

A program futása az alábbiak szerint módosult:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: A,30
Hibás az 1. adat!
Kérek két számot: 7,B
Hibás a 2. adat!
Kérek két számot:
Kérek két számot:
```

A **switch** utasításban az egyes esetekhez (**case**) tartozó utasításokat a

```
case n:
```

címke mögé kell helyezni. Azon esetek előfordulásakor, amelyeket külön nem vizsgáltunk a

```
default:
```

címke után megadott programrészlet hajtódik végre. (A példában erre a célra a **case** 2: címkét is használhattuk volna.)

Utolsó lépésként építsünk be saját ellenőrzést a ***sscanf*** függvény hívását megelőzően. Az ellenőrzés nagyon egyszerű, csak azt vizsgáljuk, hogy van-e az input szövegben nem megengedett karakter. Természetesen ez az ellenőrzés csak szükséges, de nem elégséges feltétele a helyes adatbevitelnek.

Gondoljuk át, milyen karakterek szerepelhetnek a sztringben hibátlan adatbevitel esetén:

számjegyek:	0,1,2,3,4,5,6,7,8,9
előjelek:	+,-
elválasztó:	vessző (,) és szóköz.

Ahhoz, hogy ezt az ellenőrző programrészletet később más programban is felhasználhassuk érdemes függvény formájában megvalósítani azt. Saját (nem könyvtári) függvények készítésekor az alábbi kérdéseket kell tisztáznunk:

- A függvény neve és működésének algoritmusa. (Ezt a példaprogram bemutatása után részletezzük.)
- A függvény típusa, ami annak tisztázását jelenti, hogy milyen értéket adjon vissza a nevével a függvény.
- A függvény argumentumainak szerepe és típusa.

Az alábbi példaprogramban a *Teszt* függvény paraméterként kapja meg az input sztringet, és egy kettős **for** ciklus segítségével megnézi, hogy minden egyes karaktere az input sztringnek benne van-e a minta sztringben. A külső **for** ciklus változója (*i*) a paraméterként kapott sztringen lépked karakterenként, míg a belső **for** ciklus *j* változója a minta sztringben fut körbe. (Mindkét ciklusban a leállási feltétel a sztring végét jelölő 0-ás byte elérése.)

Amennyiben eltérés jelentkezik, ezt az *ok* változó *0* értékkel jelzi és a függvény visszatér a hívás helyére. A függvény értéke ebben az esetben az *i* változó értéke lesz. Ha a ciklusok lefutottak, akkor elmondhatjuk, hogy csak megengedett karaktereket tartalmazott a paraméterként kapott sztring és a függvény értékét *-1*-re állítjuk. A program teljes listája a következő oldalon látható:


```
#include <stdio.h>
#include <conio.h>

/* Az ellenőrző függvény deklarációja */
int Teszt(char*);

main()
{
  int a,b,c;
  int poz;
  char str[160];

  do
  {
     printf("Kérek két számot: ");
     gets(str);
     if (str[0]==0) continue;
     poz=Teszt(str);                /* Az ellenőrzés elvégzése */
     if (poz!=-1)                   /* Ha van hibás karakter   */
     {
       printf("%s\n", str);         /* Az input sor kiírása    */
       printf("%*c\n",poz+1,'^'); /* A hiba helyének jelölése*/
       continue;                    /* A következő iteráció    */
     }
     switch (sscanf(str,"%d,%d",&a,&b))
     {
       case 0:
          printf("Hibás az 1. adat!\n");
          break;
       case 1:
          printf("Hibás a 2. adat!\n");
          break;
       default:
          c=a+b;
          printf("A két szám összege: %d + %d = %d\n",a,b,c);
          break;
     }
  } while (getch()!=32);
}

/* Az ellenőrző függvény definíciója */
int Teszt(char *p)
{
  static char minta[]="0123456789+-, ";   /* A jó karakterek */
  int i,j, ok;

  for (i=0; p[i]; i++) {
    ok=0;
    for (j=0; minta[j]; j++)
         if (p[i] == minta[j]) ok=1;
    if (ok==0) return i;
  }
  return -1;
}
```

A főprogramban (***main***) teszteljük a függvény által visszaadott értéket (*poz*), és amennyiben ez nem *-1*, akkor kiírjuk a képernyőre az input sort és az alatta lévő sorban a '^' karakterrel jelöljük meg a hibás karakter helyét. A '^' kiírásához használt

```
printf("%*c\n",poz+1,'^');
```

sorban a `%*c` formátum karakterek kiírására szolgál. A karakter a `poz+1` szélességű terület bal oldalára igazítva jelenik meg. A mezőszélesség az argumentumlistában is megadhatjuk, ha a formátumban a csillag (*) karakter szerepel a formátumjel (c) előtt.

Nézzük meg a program futásának eredményét:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: A,30
A,30
^
Kérek két számot: 7,B
7,B
  ^
```

A C programok fejlesztése során általában számos függvényt meg kell írnunk, azonban sokat használhatunk a gazdag szabványos könyvtárból is. Az előző ellenőrzés elvégzésére szintén találunk könyvtári függvényt a sztringkezelő függvények csoportjában. Ennek felhasználásával a program megírása könnyebbé és áttekinthetőbbé válik.

A kérdéses függvény az ***strspn***, amely megnézi, hogy az első argumentumban megadott sztringnek van-e a második argumentumban szereplő sztring karaktereitől eltérő karaktere. Amennyiben van, úgy annak pozíciója lesz a függvény értéke, de ha nincs, akkor a függvényérték az első argumentumsztring hosszát tartalmazza.

A példában a sztring hosszának lekérdezésére szintén könyvtári függvényt használtunk - ***strlen***. Nem szabad megfeledkeznünk azonban a sztringkezelő függvények deklarációját tartalmazó STRING.H állomány programunkba való beépítéséről. A könyvtári hívások felhasználásával a programunk végleges változata:


```
#include <stdio.h>
#include <conio.h>
#include <string.h>

main()
{
  int a,b,c;
  int poz;
  char str[160];

  do
  {
    printf("Kérek két számot: ");
    gets(str);
    if (str[0]==0) continue;
    poz=strspn(str,"0123456789+-, "); /* Ellenőrzés            */
    if (poz!=strlen(str))             /* Ha van hibás karakter*/
    {
       printf("%s\n", str);           /* Az input sor kiírása  */
       printf("%*c\n",poz+1,'^');     /* A hiba helye          */
       continue;                      /* A következő iteráció  */
     }
     switch (sscanf(str,"%d,%d",&a,&b))
     {
       case 0:
         printf("Hibás az 1. adat!\n");
         break;
       case 1:
         printf("Hibás a 2. adat!\n");
         break;
       default:
         c=a+b;
         printf("A két szám összege: %d + %d = %d\n",a,b,c);
         break;
     }
  } while (getch()!=32);
}
```

A program futtatásakor az előző változattal megegyező eredményt kapunk:

```
Kérek két számot: 7,30
A két szám összege: 7 + 30 = 37
Kérek két számot: A,30
A,30
^
Kérek két számot: 7,B
7,B
  ^
```

A könyv tanulmányozása során idáig eljutva, a programváltozatokat megvalósítva jó néhány működő C programot sikerült előállítani. Természetesen nem minden lépést magyaráztunk teljes részletességgel - a nem tisztázott kérdésekre a könyv további fejezetei tartalmazzák a választ. Amennyiben nem sikerült végigmenni ezen a kis gyalogúton, úgy ajánljuk, hogy a könyvünk első részének feldolgozása után lapozzon vissza ehhez a fejezethez és próbálkozzon újra.

3. A C nyelv lépésről-lépésre

Ebben a fejezetben a szabványos C nyelvvel ismertetjük meg az Olvasót. A tárgyalásnál a programozási nyelvek szokásos felépítését követjük, számos példa bemutatásával illusztrálva az elmondottakat. Az egyes részek végén található ellenőrző kérdések és feladatok megválaszolásával mindenki maga ellenőrizheti, hogy megfelelő mélységben megértette-e a fejezet anyagát.

3.1. A C nyelv alapelemei

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

3.1.1. A nyelv jelkészlete

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

```
A   B   C   D   E   F   G   H   I   J
K   L   M   N   O   P   Q   R   S   T
U   V   W   X   Y   Z

a   b   c   d   e   f   g   h   i   j
k   l   m   n   o   p   q   r   s   t
u   v   w   z   y   z

!   "   #   %   &   '   (   )   *   +
,   -   /   :   ;   <   =   >   ?   [
\   ]   ^   _   {   |   }   ~
```

A nem látható karakterek közül ide tartoznak még a szóköz, a vízszintes és függőleges tabulátor, a soremelés és a lapdobás karakterek is, melyek feladata a forrásszöveg tagolása. (Ezeket a karaktereket összefoglaló néven *white-space* karaktereknek hívjuk.)

Azok a karakterek, amelyeket nem tartalmaz a C nyelv karakterkészlete szintén szerepelhetnek a programban, de csak megjegyzések és sztring literálok (szöveg konstansok) belsejében.

3.1.2. A C nyelv azonosítói

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

```
_alap, szam1, kezdo_betu
```

A legtöbb programozási nyelvtől eltérően a C nyelv az azonosítókban megkülönbözteti a kis- és a nagybetűket. Ezért az alábbi nevek egymástól függetlenül használhatók a programban (nem azonosak):

```
xpuffer, Xpuffer, xPuffer, XPuffer, XPUFFER
```

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

```
boolean, TRUE, FALSE
```

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

```
FelsoSarok, XKoord, Y_Koord,
```

Bizonyos azonosítók speciális jelentést hordoznak. Ezeket a neveket foglalt szavaknak vagy kulcsszavaknak nevezzük. A foglalt szavakat a programban

csak a hozzájuk rendelt értelmezésnek megfelelően lehet használni. A kulcsszavakat nem lehet átdefiniálni, új jelentéssel ellátni.

Az alábbi táblázatban összefoglaltuk az ANSI C nyelv kulcsszavait:

auto	**double**	**int**	**struct**
break	**else**	**long**	**switch**
case	**enum**	**register**	**typedef**
char	**extern**	**return**	**union**
const	**float**	**short**	**unsigned**
continue	**for**	**signed**	**void**
default	**goto**	**sizeof**	**volatile**
do	**if**	**static**	**while**

A legtöbb fordítóprogram kibővíti a szabványos kulcsszavakat saját jelentéssel bíró szavakkal. Megjegyezzük, hogy ezen bővített kulcsszavak használata jelentősen csökkenti a C program hordozhatóságát. A Turbo C 2.0 fordító által használt nem szabványos foglalt szavak:

asm	**huge**	**pascal**	**_es**
cdecl	**interrupt**	**_cs**	**_ss**
far	**near**	**_ds**	

Végezetül nézzünk példát néhány hibás azonosítóra:

```
x-y, y%, Bal Also, 2ab, for
```

Az első három azonosító azért hibás, mert nem megengedett karaktereket (-. % és szóköz) tartalmaznak, a negyedik azonosító pedig számmal kezdődik. Az utolsó azonosító, a **for**, foglalt szó, ezért saját azonosítóként nem használható.

3.1.3. Konstansok

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

A C nyelvben karakteres, egész, felsorolt és lebegőpontos konstansokat használhatunk. A felsorolt (**enum**) konstansok definiálásával a típusokat ismertető fejezetben részletesen foglalkozunk. Tekintsük át az egyes konstans fajták megadási módját.

3.1.3.1. Egész konstansok

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

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

```
1994,    -1990,    32,      -1,      0
```

Oktális (8-as alapú) egész konstansok első jegye 0, amelyet oktális számjegyek (0, 1, 2, 3, 4, 5, 6, 7) követnek:

```
03712,   -03706,   040,     -01,     0
```

Hexadecimális (16-os alapú) egész konstansokat a 0x, illetve a 0X előtag különbözteti meg az előző két konstans fajtától. Az előtagot hexadecimális jegyek (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a/A, b/B, c/C, d/D, e/E, f/F) követik:

```
0x7cA,   -0X7c6,   0x20,    -0x1,    0
```

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

A fenti példákban közönséges egész számokat adtunk meg. Ha azonban előjel nélküli (**unsigned**) egészet kívánunk használni, akkor az **u** vagy az **U** betűt kell a szám után írnunk:

```
65535u,   0177777U,     0xFFFFu
```

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

```
19871207L,     0x12f35e71
```

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

```
3087007744UL,      0xB8000000LU
```

3.1.3.2. Karakter konstansok

A karakter konstansok egyszeres idézőjelek (' - aposztróf) közé zárt egy vagy több karaktert tartalmazó karaktersorozatok:

```
'a', '1', '@', 'é', 'ab', '01'
```

Az egyetlen karaktert tartalmazó karakter konstansok által képviselt számérték a karakter kódja. Több karaktert tartalmazó konstansok számértéke implementációfüggő. (Az egyértelműség kedvéért könyvünkben karakter konstans kifejezés alatt az egyetlen karaktert tartalmazó esetet értjük.)

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

Az táblázat utolsó két sorában megadott módszer segítségével tetszőleges (egy bytes-os) karaktert megadhatunk, ha ismerjük annak oktális vagy hexadecimális kódját. A C nyelvben gyakran használjuk a 0 kódú karaktert (NUL, sztring vége jel), amit egyszerűen '\0' alakban adhatunk meg.

Értelmezés	ASCII karakter	Escape szekvencia
csengõ	BEL	'\a'
visszatörlés	BS	'\b'
lapdobás	FF	'\f'
újsor	NL (LF)	'\n'
kocsivissza	CR	'\r'
vízszintes tabulálás	HT	'\t'
függõleges tabulálás	VT	'\v'
aposztróf	'	'\''
idézõjel	"	'\"'
backslash	\	'\\'
kérdõjel	?	'\?'
ASCII karakter oktális kóddal megadva	ooo	'\ooo'
ASCII karakter hexadecimális kóddal megadva	hh	'\xhh'

Az elmondottak figyelembe vételével nézzük meg, hogyan tudjuk a C programban megadni a nagy *A* betűt. A hivatkozáshoz egyaránt használhatjuk az egész és a karakter konstansok előállítási formáit:

```
65,      0101,    0x41
'A',     '\101'   '\x41'
```

3.1.3.3. Lebegőpontos konstansok

A lebegőpontos konstans olyan decimális szám, amely előjeles valós számot reprezentál. A valós szám általában egészrészből, tizedes törtrészből és kitevőből tevődik össze. Az egész- és törtrészt tizedespont (.) kapcsolja össze, míg a kitevő (10 hatványkitevője) az **e**, vagy az **E** betűt követi. Az egyes részek értelemszerűen el is hagyhatók, mint ahogy az alábbi példákból is látható:

C nyelv	Matematikai értelmezés
.1	0,1
-2.	-2,0
100.45	100,45
2e-3	0,002
11E2	1100,0
-3.1415925	-3,1415925
31415925E-7	3,1415925

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

```
3.1415F, 2.7182f
```

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

```
3.14159265358979323851,     2.71828182845904523541
```

3.1.4. Sztring literálok

A sztring literál, amit sztring konstansnak is szoktak hívni, kettős idézőjelek közé zárt karaktersorozatot jelent:

```
"Ez egy sztring konstans!"
```

A megadott karaktersorozatot a statikus memóriaterületen helyezi el a fordító, és ugyancsak eltárolja a sztringet záró '\0' karaktert (nullás byte-ot) is A sztring konstans tartalmazhat *escape* szekvenciákat is,

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

amelyek esetén csak a nekik megfelelő karakter (egy byte) kerül tárolásra.

Hosszú szövegeket a *backslash* (\) karakter segítségével több programsorban is elhelyezhetünk,

```
"Hosszú szöveget két vagy több darab\
ra tördelhetünk."
```

azonban a memóriában egyetlen sztring konstansként kerül tárolásra:

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

Egymás után elhelyezkedő sztring konstansokat szintén egyetlen sztring literálként tárolja a fordító. A fentivel megegyező eredményt kapunk, ha a programban az alábbi sor szerepel:

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

(A sztring literál maximális hossza implementációfüggő - az ANSI C szabvány legalább 590 karakteres hosszat javasol.)

3.1.5. Megjegyzések

A megjegyzés olyan karaktersorozat, amelyet a /* (perjel és csillag) karakterek előznek meg és a */ (csillag és perjel) karakterek zárnak.

```
/* Ez egy jó megjegyzés */
```

Hosszabb megjegyzéseket több sorban is elhelyezhetünk:

```
/* Ez is jó, mivel a megjegyzés
   több sorban is folytatható!
*/
```

A megjegyzések célja, hogy a program forráskódja jól dokumentált, ezáltal egyszerűen értelmezhető, jól olvasható legyen.

Programfile-ok elején és függvények előtt, általában nagyobb megjegyzésblokkban ismertetjük az adott program, alprogram használatához szükséges információkat. Ehhez jól elkülönülő megjegyzést szokás írni, mint például a következő:

```
/*****************************************
 * A program neve              :
 * A program készítője         :
 * Az utolsó változtatás ideje:
 *****************************************/
```


3.1.6. Operátorok

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

```
!     !=    %     %=    &     &&    &=    ()    *     *=
+     ++    +=    ,     -     --    -=    ->    .     /
/=    <     <=    <<    <<=   =     ==    >     >=    >>
>>=   ?:    []    ^     ^=          sizeof    |     |=
||    ~
```

3.1.7. Írásjelek

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

Írásjel	Az írásjel szerepe
[]	Tömb kijelölése, méretének megadása,
()	A paraméter- és az argumentum lista kijelölése,
{}	Kódblokk vagy függvény behatárolása,
*	A mutató típus jelölése a deklarációkban,
,	A függvény argumentumok elválasztása,
:	Címke elválasztása
;	Az utasítás végének jelölése
...	Változó hosszúságú argumentumlista jelölése,
#	Előfordító direktíva jelölése.

Ellenőrző kérdések:

1 Milyen karakterkészletet használ a C nyelv?
2 Mely szimbólumoknál nem lehet szóközt tenni?
3 Mik azok a foglalt szavak?

4. Érzékenyek-e a foglalt szavak a kis- és a nagybetűk használatára?
5. Mit nevezünk azonosítóknak?
6. Érzékenyek-e a C nyelv azonosítói a kis- és nagybetűk használatára?
7. Válasszuk ki az alábbi azonosítók közül a hibásan megadott azonosítókat!

```
sor   Sor   SOR   s1r   s/p   a_1   a*1   z#2   C2   2a
for   b     h%
```

8. Hogyan osztályozzuk a számkonstansokat ?
9. Mi a szerepe az *E*, illetve az *e* betűnek az alábbi számkonstansokban?

```
2.014E+3      -1.5647e-2
```

10. Hogyan adunk meg oktális számot?
11. Hogyan jelöljük a hexadecimális számokat?
12. Hogyan adunk meg karakteres konstanst?
13. Hogyan adjuk meg a sztring konstansokat?
14. Melyek a megjegyzés írásának a szabályai?

3.2. A C program szerkezete

A C nyelven megírt program egy vagy több forrás file-ban (fordítási egységben, modulban) helyezkedik el, melyek kiterjesztése .C A programhoz általában ún. *include* (*header*) file-ok is csatlakoznak, melyeket az **#include** előfordító utasítás segítségével beépítünk a forrásállományokba. Az *include* file-ok szokásos kiterjesztése **.H.**

A C program részeinek részletes bemutatása előtt, nézzünk meg néhány egyszerű C programot.

3.2.1. A legegyszerűbb C program

A legegyszerűbb C program egyetlen sorból áll:

```
main() { }
```

Ez az egysoros program egyetlen függvényt tartalmaz, amelyet kötelezően ***main***-nek kell nevezni. A függvény nevét követően zárójelek () fogják közre a paramétereket. Ebben az esetben nincsenek paraméterek, de a zárójeleket ki kell tenni. A { } (kapcsos) zárójelek fogják össze a függvény törzsét képező blokkot. A példában a függvény törzse nem tartalmaz utasításokat.

3.2.2. Egy szöveget kiíró C program

A szöveg kiírását végző C program, már több sorból áll:

```
#include <stdio.h>

main()
{
  printf("C program\n");
}
```

A ***main*** függvény törzse egyetlen utasítást, egy függvényhívást, tartalmaz. A ***printf*** könyvtári függvény segítségével írjuk ki az idézőjelek között megadott szöveget. A ***printf*** függvény leírását (deklarációját) az STDIO.H *include* file tartalmazza, amit emiatt beépítettünk a programunkba.

3.2.3. Egyetlen modulból felépülő C program

A C nyelv a funkció-orientált programozási módszert támogatja. Ez a módszer azt jelenti, hogy valamely probléma megoldásához különböző funkciókat (műveleteket) megvalósító programegységek (alprogramok) készítésével jutunk el. A C nyelvben ezeket a programegységeket függvényeknek hívjuk.

A megírt függvényeket akár a ***main*** függvénnyel azonos forrás file-ban, de tetszőleges számú más forrásállományban is elhelyezhetjük. A ***main*** függvény kitüntetett szerepe abban áll, hogy kijelöli a program belépési pontját, vagyis a program futása a ***main*** függvény indításával kezdődik. Ezért érthető az a megkötés, hogy minden C programnak tartalmaznia kell egy ***main*** nevű függvényt, de csak egy példányban

Az alábbi példában egyetlen forrás file-ból álló C program szerkezete követhető nyomon:

```
#include <stdio.h>          /*Előfordító utasítások       */
#define  EGY   1
#define  KETTO 2

int sum(int, int);          /*Globális definíciók és      */
int e;                      /*deklarációk                 */

/*A main függvény definíciója*/
main()                      {
  int a,b;                  /*Lokális definíciók és       */
                            /*deklarációk                 */

  a=EGY; b=KETTO;           /*Utasítások                  */
  e=sum(a,b);
  printf("Az összeg: %d\n",e);
}

/*A sum függvény definíciója */
int sum(int x, int y)       {
  int z;                    /*Lokális definíciók és       */
                            /*deklarációk                 */

  z=x+y;                    /*Utasítások                  */
  return z;
}
```


A fenti példaprogram két függvény tartalmaz. A *sum* függvény a paraméterként kapott két számot összeadja és függvényértékként ezt az összeget szolgáltatja. A ***main*** függvényben az *1*-et és a *2*-őt összegezzük majd az eredményt kiírjuk a képernyőre:

```
Az összeg:  3
```

A függvények törzsét képező blokkon belül először a deklarációkat és definíciókat kell elhelyeznünk, amelyeket végrehajtandó utasítások követnek. Az utasítások írása során gyakran használunk olyan blokkokat, amelyek további utasításokat tartalmaznak, a deklarációs rész elhagyásával.

Tároljuk a példaprogramot a CPROG.C állományban. A futtatható file előállítása (*translation*) két fő lépésben megy végbe:

- Az első lépés a CPROG.C forrás file fordítása (*compiling*), melynek során olyan közbenső állomány jön létre, amely a program adatait és gépi szintű utasításait tartalmazza (IBM PC számítógépen ezt a file-t tárgymodulnak (*object modul*) hívjuk, és .OBJ kiterjesztésű állományban helyezkedik el.) A fordítás eredménye még nem futtatható, hiszen tartalmazhat olyan (pl könyvtári) hivatkozásokat, amelyek még nem kerültek feloldásra.

- A második lépés feladata a futtatható állomány összeállítása (*linking*). Itt fontos szerepet játszanak a C szabványos függvények kódját tartalmazó könyvtárak, melyek általában LIB kiterjesztésű állományokban helyezkednek el. (IBM PC számítógépen a keletkező futtatható programot EXE file tartalmazza.)

3.2.4. Több modulból álló C program

Több modul használata nagy programok esetén szükségszerű, hisz a fordítók nem képesek akármekkora forrásállományt feldolgozni. Ez a lehetőség azonban ennél jóval többet jelent. A C nyelv tartalmaz eszközöket a moduláris programozás elvének megvalósításához. A moduláris programozás lényege, hogy minden modul önálló fordítási egységet képez, melyeknél érvényesül az adatrejtés elve.

Mivel a modulok külön-külön lefordíthatók, nagy program fejlesztése, javítása esetén nem szükséges minden modult újrafordítani. Ez a megoldás jelentősen csökkenti a futtatható program előállításának idejét.

Az adatrejtés elvét a későbbiekben tárgyalásra kerülő file-szintű érvényességi tartomány (láthatóság, *scope*) és a tárolási osztályok biztosítják. Ezek megfelelő használatával a modul bizonyos nevei kívülről (*extern*) is láthatók lesznek, míg a többi név elérhetősége a modulra korlátozódik.

A több modulból álló C program fordításának bemutatásához az előző alfejezet példáját vágjuk ketté. A CPROG1.C file csak a ***main*** függvényt és a CPROG2.C állományban elhelyezkedő *sum* függvény leírását (prototípusát) tartalmazza. Az **extern** kulcsszó jelzi, hogy a *sum* függvényt más modulban kell a szerkesztőnek keresnie.

```
#include <stdio.h>              /*Előfordító utasítások    */
#define  EGY   1
#define  KETTO 2

extern int sum(int, int);       /*Globális definíciók és   */
int e;                          /*deklarációk              */

/*A main függvény definíciója*/
main()
{
  int a,b;                      /*Lokális definíciók és    */
                                /*deklarációk              */

  a=EGY; b=KETTO;               /*Utasítások               */
  e=sum(a,b);
  printf("Az összeg: %d\n",e);
}
```

A CPROG2.C file-ban csak a *sum* függvény található:

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

  z=x+y;                        /*Utasítások               */
  return z;
}
```


A futtatható file előállításakor minden modult külön le kell fordítanunk, (CPROG1.OBJ, CPROG2.OBJ). A keletkező tárgymodulokat a szerkesztő (*linker*) építi össze a megfelelő könyvtári hivatkozások feloldásával. (A Turbo C 2.0-ban ún. *project* file írásával a fenti lépéseket automatikusan elvégzi az integrált fejlesztői környezet.)

Általában elmondható, hogy a C forrásprogram előfordító utasítások, deklarációk és definíciók, utasításblokkok és függvények kollekciója. Az egyes részek további tagolását és részletes ismertetését a következő fejezetekben végezzük el.

Ellenőrző kérdések

1 Milyen nevű függvénynek kell kötelezően szerepelni a C programban?
2 Milyen típusú zárójelbe kell tenni a függvény paramétereit?
4 Milyen típusú zárójelbe kell tenni a függvény törzsét?
5 Hogy néz ki egy általános C program szerkezete?

3.3. Típusok, változók, konstansok

Mint minden programozási nyelvben - így a C-ben is - minden felhasznált névről meg kell mondanunk, hogy mi az és hogy mire szeretnénk használni. Enélkül a fordító általában nem tud mit kezdeni az adott névvel.

A C nyelv szóhasználatával élve mindig deklarálnunk kell az általunk alkalmazni kívánt neveket. A deklaráció (leírás) során csak a név tulajdonságait (típus, tárolási osztály, láthatóság, stb.) közöljük a fordítóval. Ha azonban azt szeretnénk, hogy az adott deklarációnak megfelelő objektum is létrejöjjön a memóriában, akkor definíciót kell használnunk. (C-ben az objektum kifejezés olyan memóriaterületet jelöl, amely egy vagy több értéket tartalmaz.) A definíció tehát olyan deklaráció, amely helyfoglalással jár. Ugyanazt az objektumot többször is deklarálhatjuk, azonban az egymás követő deklarációknak azonosnak kell lenniük. Ezzel szemben valamely objektum definíciója csak egyetlen egyszer szerepelhet a programban.

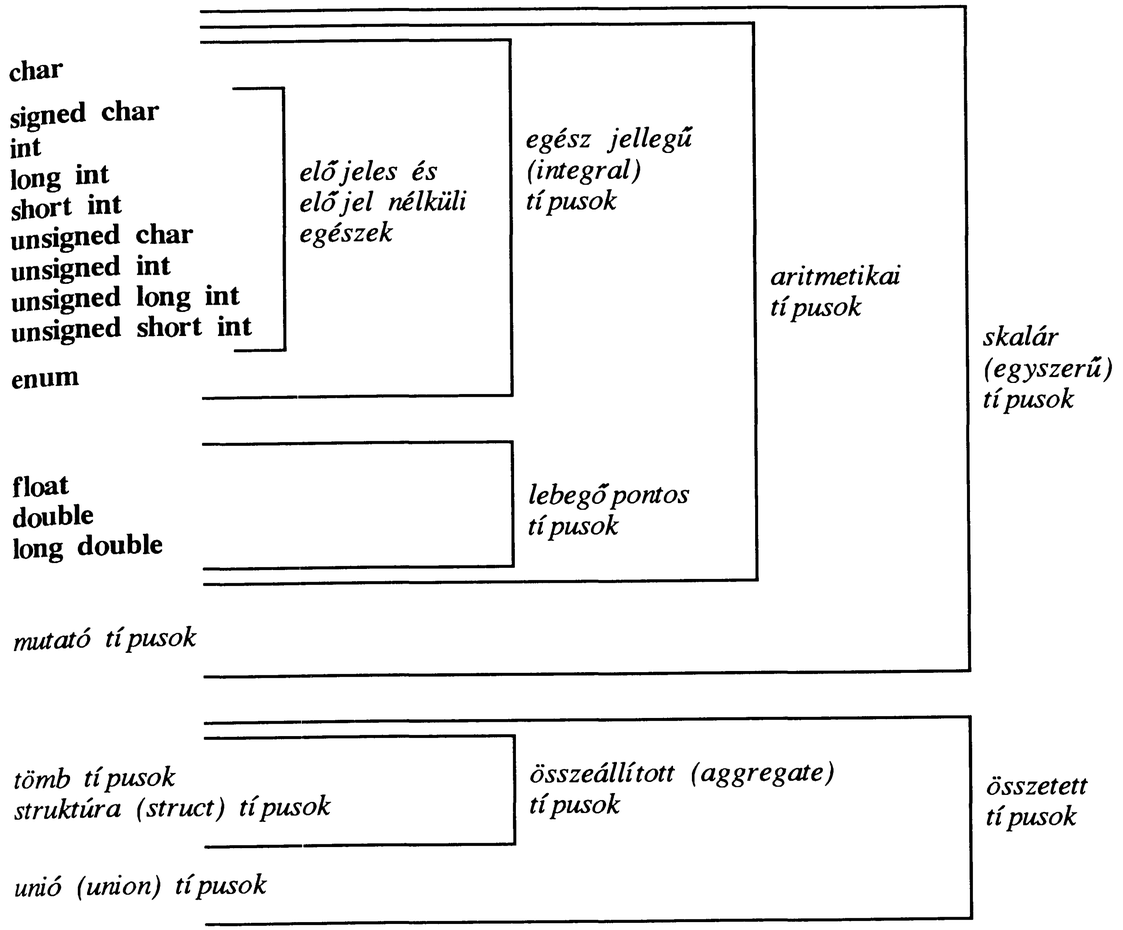
A definíció és a deklaráció közötti különbséget nemcsak a gyakori használatuk miatt emeltük ki, hanem azért is, mivel a programozók sokszor összekeverik ezt a két fogalmat, és az egyiket a másiknak megfelelő tartalommal használják. Kezdő programozó számára hasznos segítséget jelent, ha a későbbiekben a program írása során mindig tisztázza, hogy most éppen deklarál vagy definiál valamilyen objektumot.

A fordító számára a deklaráció (definíció) során közölt egyik legfontosabb információ a típus.

3.3.1. A C nyelv típusai

A típus az objektumban tárolt - vagy függvény által visszaadott - érték értelmezését határozza meg. A C szabvány különböző szempontok alapján csoportosítja a típusokat. Nézzünk ezek közül néhány megközelítést.

Az egyik legteljesebb csoportosítás a 3.1. ábrán látható, ahol felsorolásban minden típusnév szerepel.



3 1 ábra
A C-típusok csoportosítása

Az előzőeknél sokkal egyszerűbb, azonban bizonyos szempontból kibővített értelmezését jelenti a típusoknak az a csoportosítás amely az alaptípusokat (*base types*) és a származtatott (*derived*) típusokat különbözteti meg.

Az alaptípusokhoz a **char**, az előjeles és előjel nélküli egészek és a lebegőpontos típusok tartoznak (ez nem más, mint az aritmetikai típusok csoportja az **enum** típus nélkül).

A származtatott típusok csoportja az alaptípusok felhasználásával felépített tömb, függvény, mutató, struktúra és unió típusokat tartalmazza.

3.3.1.1. Típuselőírások, típusmódosítók

A típusnevek felépítése során bizonyos kulcsszavak megengedett kombinációját használjuk. A 3.1. ábrán is látható, hogy a C nyelvben viszonylag kevés az olyan - egyetlen kulcsszót tartalmazó - típusnév, amely nem szerepel más kulcsszavak előtt:

char
int
float
double
enum
struct
union

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

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

A **short int** típus egy rövidebb (2 byte-on tárolt), míg a **long int** típus egy hosszabb (4 byte-on tárolt) egész típust definiál. Az ANSI C szabványban bevezetett **long double** típus az adott számítógépen értelmezett legnagyobb pontosságú lebegőpontos típust jelöli.

Az egész típusok lehetnek előjelesek (*signed*) és előjel nélküliek (*unsigned*). Az **int** típusok (**int**, **short int**, **long int**) alapértelmezés szerint pozitív és negatív egészek tárolására egyaránt alkalmasak, míg a **char** típus előjelének értelmezése implementációfüggő. A fenti négy típus előjeles vagy előjel nélküli volta egyértelművé tehető a **signed**, illetve az **unsigned** típusmódosítók megadásával.

Ezen megoldások bevezetése elsősorban azzal magyarázható, hogy a **char** és az **int** típusok értelmezése függ a C nyelv implementációjától (a hardvertől,

és az operációs rendszertől). A típusmódosítókat felhasználva, adott a lehetőség olyan programok írására, amelyek ugyanúgy futnak a különböző típusú számítógépeken.

A típusmódosítók napjainkra szerves részévé váltak a C nyelvnek, hiszen önmagukban típuselőírásként is használhatók. Az alábbiakban (abc-sorrendben) összefoglaltuk a lehetséges típuselőírásokat. Az előírások soronként azonos típusokat jelölnek.

char
double
enum *típusnév*
float
int, **signed**, **signed int**
long double
long int, **long**, **signed long**, **signed long int**
signed char
short int, **short**, **signed short**, **signed short int**
struct *típusnév*
union *típusnév*
unsigned char
unsigned int, **unsigned**
unsigned long, **unsigned long int**
unsigned short, **unsigned short int**
void

3.3.1.2. Típusminősítők

A típuselőírásokat típusminősítővel együtt használva a deklarált azonosítóhoz az alábbi két tulajdonság egyikét rendelhetjük. A **const** kulcsszóval olyan objektumot deklarálhatunk, amely nem változtatható meg (csak olvasható objektum).

A **volatile** típusminősítővel olyan objektum hozható létre, amelyet (teljesen hivatalosan) a programtól független kód (például egy másik futó folyamat, vagy megszakítást kezelő rutin) is megváltoztathat. A **volatile** azt közli a fordítóval, hogy az nem tud mindent, ami az adott objektummal történhet. (Ezért például a fordító minden egyes, ilyen tulajdonságú objektumra történő hivatkozáskor, a memóriából veszi fel az objektumhoz tartozó értéket.)

Mindkét deklaráció lehetőséget biztosít a fordítóprogramok számára bizonyos kódszintű optimálizációk beépítésére. Például a **const** objektum esetén szükségtelen azt ellenőrizni, hogy az megváltozott-e. Nézzünk néhány példát a típusminősítők használatára:

int const
const int
volatile char
long int volatile

Ebben a fejezetben csak az aritmetikai típusok ismertetésére szorítkozunk. A származtatott típusok bemutatásával könyvünk további fejezetei foglalkoznak.

3.3.1.3. A karakter típus

A **char** típusú adatelem az adatok legkisebb címezhető egységét jelöli. Ez az adatelem a memória egyetlen byte-ját tartalmazza, és egyben alkalmas a C karakterkészlet karaktereinek tárolására.

A **char** típus - a nyelv implementációjától függően - alapértelmezés szerint lehet előjeles vagy előjel nélküli. A karakteres típusok értékkészletét a Turbo C 2.0-és rendszernek megfelelően a 3 2 ábrán foglaltuk össze. (A Turbo C alaphelyzetben előjeles **char** típust használ, ez azonban fordítási opcióban átállítható.)

A szabványnak megfelelően a LIMITS H *include* file előfordító makrokat definiál az értékkészlet határainak elérésére A **char** típus értékei a ***CHAR_MIN*** és a ***CHAR_MAX*** határok között helyezkednek el. Külön makrok adják meg a **signed char** típushoz tartozó (***SCHAR_MIN***, ***SCHAR_MAX***) határokat, illetve az **unsigned char** típushoz tartozó maximális értéket (***UCHAR_MAX***).

3.3.1.4. Egész típusok

Az egész típusok alapját az **int** típus képezi, melynek mérete általában megegyezik a számítógépi szó (regiszter) méretével, vagyis hardverfüggő. A szabvány megengedi ezt a bizonytalanságot és csak annyit köt ki, hogy az **int**

típus nem lehet kisebb a **short** típusnál és nem lehet nagyobb a **long** típusnál:

short ≤ **int** ≤ **long**

Az IBM PC számítógépeken az MS-DOS alatt működő C fordítók többsége 16 bites (rövid) egészként definiálja az **int** típust, míg az újabb 32-bites fordítók 4 byte (hosszú) **int** típust használnak. A Turbo C 2.0 rendszer alatti egész típusok értékhatárait és méreteit a 3.2. ábra tartalmazza.

A LIMITS.H az egész típusokhoz szintén definiál szabványos makrokat, amelyek segítségével lekérdezhetjük az értékhatárokat:

Típus	Értékhatárok	
	alsó	felső
`short int`	*SHRT_MIN*	*SHRT_MAX*
`unsigned short int`	*0*	*USHRT_MAX*
`int`	*INT_MIN*	*INT_MAX*
`unsigned int`	*0*	*UINT_MAX*
`long int`	*LONG_MIN*	*LONG_MAX*
`unsigned long int`	*0*	*ULONG_MAX*

3.3.1.5. A felsorolt típus

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

```
enum azonosító { felsorolás }
```

A felsorolásban szereplő konstansok **int** típusúak lesznek. Nézzünk néhány példát a felsorolt típus létrehozására:

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

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

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

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

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

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

A **enum** típust elsősorban csoportos konstansdefiniálásra használjuk. Ellentétben az erősen típusos nyelvekkel (mint például a Pascal) a C nyelv semmilyen ellenőrzést nem végez a felsorolt típusú objektumok értékére vonatkozóan. (Egyetlen megkötés, hogy az **int** típusú legyen. Ezért az **enum** típusokat az egész típus speciális fajtájaként kezeljük.) Konstansok definiálásakor a típus nevét el is hagyhatjuk:

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

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

3.3.1.6. A lebegőpontos típusok

A **float** adattípust egyszeres pontosságú lebegőpontos számok tárolására használjuk. A **double** típus a kétszeres pontosságú lebegőpontos számokat reprezentálja. A **long double** típus az adott számítógépes környezet legnagyobb pontosságú lebegőpontos típusát hivatott lefedni. A három típus mérete között az alábbi összefüggésnek kell fennállni:

float ≤ **double** ≤ **long double**

(Elképzelhető olyan C implementáció is, ahol mind a három típus ugyanazt a tárolási formát jelöli.)

A lebegőpontos számok tárolási formátuma három részre osztható. Az első rész a szám előjelét meghatározó előjelbit, ezt követi az exponens rész, majd pedig a mantissza bitek következnek. A lebegőpontos formátummal kapcsolatban a FLOAT.H *include* file tartalmaz makrokat. A **float** típushoz kapcsolódó makrok ***FLT_***, a **double** típushoz kapcsolódók ***DBL_***, míg a **long double** típus makroi ***LDBL_*** előtaggal rendelkeznek.

A Turbo C 2.0 rendszer által használt lebegőpontos típusok számszerűsített határait szintén a 3.2. ábra tartalmazza.

Adattípus	Értékkészlet	Méret (byte)	Pontosság (jegy)
char	-128..127	1	
unsigned char	0..255	1	
signed char	-128..127	1	
int	-32768..32767	2	
unsigned int	0..65535	2	
short int	-32768..32767	2	
unsigned short	0..65535	2	
long int	-2147483648..2147483647	4	
unsigned long	0..4294967295	4	
float	3.4E-38..3.8E+38	4	6
double	1.7E-308..1.7E+308	8	15
long double	3.4E-4932..3.4E+4932	10	19

3 2 ábra
A Turbo C 2 0 rendszer alaptípusai

3.3.2. Egyszerű változók definiálása

A C program objektumait névvel látjuk el, hogy tudjunk rájuk hivatkozni. A név segítségével az objektumhoz értéket rendelhetünk, illetve lekérdezhetjük az objektumban tárolt értéket. A névvel ellátott objektumokat a programozási nyelvekben változónak szokás nevezni.

Ebben a részben az egyszerű változók definiálásával ismerkedünk meg. Az egyszerű változó jellemzője, hogy csak egyetlen egész jellegű (*integral*) vagy lebegőpontos érték tárolására alkalmas. Az itt megismert megoldások azonban segítségünkre lesznek a származtatott típusú változók felhasználásakor is.

Nézzük először általános formában, hogyan néz ki a változók definíciója (deklarációja):

⟨*tárolási osztály*⟩ *típus* ⟨*típus ...*⟩ *változónév*⟨=*kezdőérték*⟩ ⟨, ...⟩,

Az általánosított formában a ⟨ ⟩ jelek az opcionálisan megadható részeket jelölik, míg a három pont az előző definíciós elem ismételhetőségére utal. Sokkal érthetőbb lesz a dolog, ha példákon keresztül mutatjuk be a lehetőségeket

```
int alfa;
int beta=4;
int gamma;
```

A fenti példában három egész típusú változót definiáltunk (*alfa*, *beta*, *gamma*), és a *beta* változót kezdőértékkel is elláttuk (4). Azonos típusok használata esetén a változókat vesszővel tagolva egymás után is elhelyezhetjük:

```
int alfa, beta=4, gamma;
```

Külön felhívjuk a figyelmet arra, hogy a deklarációs sort pontosvesszővel kell lezárni.

A deklarációban a típust megelőzheti néhány alapszó (**auto**, **register**, **static** vagy **extern**), amelyek az objektum tárolásával kapcsolatban tartalmaznak előírásokat. Az előírások, amiket tárolási osztálynak nevezünk, meghatározzák az objektum elhelyezkedését, láthatóságát és élettartamát. Minden változóhoz tartozik tárolási osztály, még akkor is, ha azt külön nem adtuk meg. Ha a változót a függvényeken kívül definiáljuk, akkor az alapértelmezés szerint globális (más modulból elérhető) tárolási osztállyal rendelkezik, míg a függvényeken belül definiált változók alaphelyzetben automatikus (**auto**) változók. A tárolási osztályokkal teljes részletességgel a 3.9. fejezetben foglalkozunk.

3.3.3. Saját típusok előállítása

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

```
volatile unsigned long int
```

típus. Definiáljunk a fenti típus felhasználásával egy kezdőérték nélküli változót:

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

Ha ezt a típust a program több pontján kívánjuk alkalmazni, akkor elég fáradságos mindig ezt a hosszú nevet megadni. A C nyelv tartalmaz egy speciális tárolási osztályt (**typedef**), amely lehetővé teszi, hogy érvényes típusokhoz szinonim nevet rendeljünk. A fenti példánál maradva, legyen az új típus neve *tido*:

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

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

```
tido idozites;
```

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

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

3.3.4. Konstansok a C nyelvben

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

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

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

```
const int a=30;
a=7;                    /* Hibajelzést kapunk a fordítótól. */
```

Ez a megoldás tetszőleges típusú változó esetén használható. A fordító a **const** objektumokat szintén a memóriában tárolja, így mutatók segítségével - indirekt módon - azok értéke megváltoztatható.

A másik, szintén gyakran használt megoldás, amikor az előfordító **#define** utasításával létrehozott makrok hordoznak konstans értékeket. Az előfordító használatát a 3.10 fejezet részletesen ismerteti, annyit azonban nézzünk meg, hogyan definiálhatunk konstans makrokat. Az előfordító által használt neveket csupa nagybetűvel szokás írni:

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

main()
{
     int a=KEK;
     double f;

     a=a+PIROS;
     f=90*PI/180;
}
```

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

```
main()
{
     int a=1;
     double f;

     a=a+4;
     f=90*3.14159265/180;
}
```


A harmadik lehetőség, ami a 3.3.1.5. alfejezetben ismertetett **enum** típus használatát jelenti, csak egész (**int**) típusú konstansok esetén alkalmazható. Az előző példában szereplő színkonstansokat az alábbi alakban is előállíthatjuk:

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

main()
{
     int a=kek;

     a=a+piros;
}
```

Nézzük meg, miben rejlik ezen harmadik megoldás előnye az első két módszerhez képest. Az **enum** konstansok igazi konstansok, hisz nem tárolja őket a memóriában a fordító. Míg a **#define** konstansok a definiálás helyétől a file végéig fejtik hatásukat, addig az **enum** konstansokra a szokásos C láthatósági és élettartam szabályok érvényesek.

3.3.5. Értékek, címek és mutatók

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

```
objektum = érték;
```

A C nyelven az értékadás operátor és a fenti utasítás valójában egy kifejezés, amit a fordítóprogram értékel ki. Az értékadás operátorának bal- és jobb oldalán egyaránt szerepelhetnek kifejezések, melyek azonban lényegileg különböznek egymástól. A baloldalon szereplő kifejezés azt az objektumot jelöli ki a memóriában (megcímezi), ahova a jobb oldalon megadott kifejezés értékét be kell tölteni.

3.3.5.1. Balérték és jobbérték

A fenti alakból kiindulva a C nyelv külön nevet ad a kétfajta kifejezésnek. Annak a kifejezésnek az értéke, amely az egyenlőségjel bal oldalán áll, a balérték (*lvalue*), míg a jobboldalon szereplő kifejezés értéke a jobbérték (*rvalue*).

Vegyünk példaként két egyszerű értékadást:

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

Az első értékadás során az *a* változó maga mint balérték szerepel, vagyis a változó címe jelöli ki azt az objektumot, ahova a jobboldalon megadott konstans értéket be kell másolni. A második értékadás során az *a* változó az értékadás mindkét oldalán szerepel. A baloldalon álló *a* ugyancsak az objektumot jelöli ki a memóriában (*lvalue*), míg a jobboldalon álló *a* egy jobbérték kifejezésben szerepel, melynek értékét (13) a fordító meghatározza az értékadás elvégzése előtt. Hasonló módon értelmezhető az

```
a = a;
```

utasítás is.

3.3.5.2. Ismerkedés a mutatókkal

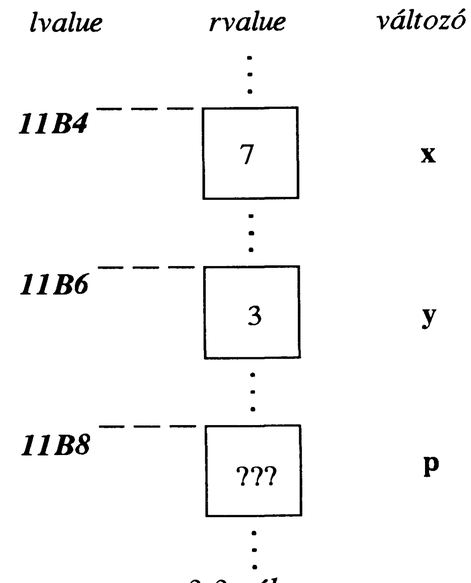
Más programozási nyelveken az értékadás hasonló gondolatmenettel megy végbe, azonban ott külön elnevezéssel nem hangsúlyozzák ki a két oldal közötti különbséget. Mi az oka annak, hogy a C nyelven különösen nagy hangsúlyt fektetünk a bal- és a jobbérték fogalmának tisztázására?

Egyik lényeges eltérés más nyelvekhez képest, hogy az értékadás művelet (operátor), nem pedig külön utasítás. A másik fontos különbség abban rejlik, hogy a C nyelvben kiemelt szereppel rendelkezik egy olyan speciális változó, amely más objektumok címét tárolja - ez a mutató (pointer). Ezzel kapcsolatban az értékadás bal oldalán szerepeltethetünk olyan kifejezéseket is, amelyek balérték, vagyis cím jellegű kifejezések.

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

Tekintsük az alábbi kis programot, amelyben két egész típusú változót definiálunk kezdőérték • megadásával, és egy pointer változót kezdőérték nélkül.


```
#include <stdio.h>

main()
{                              /* definíciók */
  int x=7, y=3;
  int *p;
                               /* a program első része */
  p=&x;
  *p=*p+x+y;
  printf("x=%d\n",x);
                              /* a program második része */
  p=&y;
  *p=x+y;
  printf("y=%d\n",y);
}
```

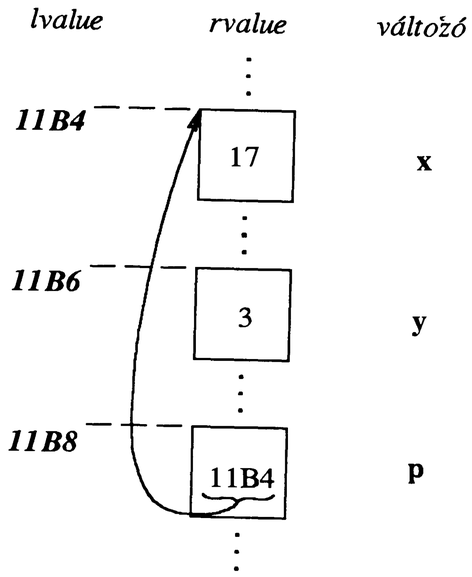


3 3 ábra
A definiált x, y és p változók

A mutató definiálásakor a mutató nevét a csillag (*) előzi meg. Mivel a p mutatónak nem adtunk kezdőértéket, az sehova se (illetve nem tudjuk, hogy hova) mutat. A definíció során létrejött három objektum a memóriában, mint ahogy az a 3.3. ábrán látható.

A program első utasításával értéket adunk a *p* mutatónak, a "címe" operátor (&) felhasználásával. Vagyis a *p* értékként kapja az *x* változó címét, aminek következtében a *p* az *x*-re fog mutatni (3.4. ábra).

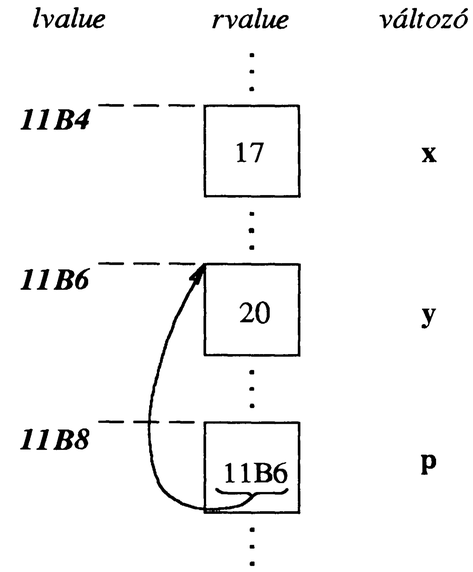
A mutatókkal kapcsolatos másik fontos művelet a "mutatott" objektumra való hivatkozás, amihez az indirektség operátorát (*) kell megadnunk a mutató előtt. A **p* kifejezés a második utasításban az *x* változót helyettesíti, hiszen a *p* az *x*-re mutató pointer, melynek következtében a mutatott objektum (a **p*) maga az *x* változó. A harmadik utasításban kiíratva az x értékét 17-et kapunk (*x=7+7+3*).



3 4 ábra

A változók értéke a program első részének végrehajtása után

A fentiekhez hasonló lépéseket használunk a program második részében (az utolsó három utasításában) is, melyek értelmezését az Olvasóra bízzuk. Segítségül felhívjuk a figyelmet a 3.5. ábrára, melyen jól megfigyelhető a változók értékének további alakulása.



3 5 ábra

A változók értéke a program második részének végrehajtása után

3.3.5.3. Mutatók és a dinamikus memóriahasználat

A fenti kis programból is kitűnnek a mutatók felhasználásának lehetőségei. Azonban meg kell jegyeznünk, hogy általában nem azért alkalmazunk mutatókat, hogy más létező objektumra mutassunk vele. Nagyon fontos lehetősége a C nyelvnek az ún. dinamikus memóriahasználat. Ennek során memóriablokkot foglalhatunk le, amelyet mutató segítségével elérhetünk, és ha már nincs szükségünk a lefoglalt területre, felszabadíthatjuk azt. A memóriafoglalás és -felszabadítás művelete (az alap I/O-hoz hasonlóan) szintén szabványos könyvtári függvények formájában van jelen a C nyelvben.

Minden memóriafoglalási kísérlet (***malloc***) után meg kell vizsgálnunk, hogy sikerült-e lefoglalni a kívánt méretű memóriablokkot. Amennyiben ezt a vizsgálatot elmulasztjuk, illetve ha kezdőérték nélküli mutatót használunk, akkor minden esélyünk megvan arra, hogy a program futása megszakadjon, vagy mint például az MS-DOS alatt a számítógépünk "lefagyjon".

Mutatók használata a "veszélyes üzem" kategóriába tartozik, azonban szakavatott kezekben a lehetőségek gaždag tárházát nyitja meg a programozó előtt.

A dinamikus memóriahasználat lépéseinek bemutatására tekintsünk egy kis programot, amelyben az *x* és *y* változók értékét összeadjuk, és egy dinamikus memóriafoglalású változóban tároljuk.

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

main()
{
  int x=7, y=3;
  int *p;

  /* int típus tárolására alkalmas terület lefoglalása */
  /* a műveletet eredményességének ellenőrzésével      */
  p=(int *)malloc(sizeof(int));
  if (p==NULL) exit(-1);

  /* Az összeg tárolása és kiírása                     */
  *p=x+y;
  printf("Az összeg: %d\n",*p);

  /* A lefoglalt terület felszabadítása */
  free(p);
}
```

*3.3.5.4. A void * típusú mutatók*

Az előző példákban pointerekkel mutattunk **int** típusú objektumokra. Az objektum eléréséhez nem elegendő az objektum címét tárolnunk (ez van a mutatóban), hanem definiálnunk kell az objektum méretét is, amit a mutató típusa közvetít a fordítónak. A C nyelv lehetővé teszi típus nélküli, ún. általános mutató használatát is:

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

amely azonban sohasem jelöl ki memóriaobjektumot. Ha ilyen mutatóval szeretnénk a hivatkozott objektumnak értéket adni, akkor felhasználói típuskonverzióval (*cast*) típust kell rendelnünk a cím mellé:


```
*(int *)ptr=5;
```

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

3.3.5.5. Többszörös indirektségű mutatók

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

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

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

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

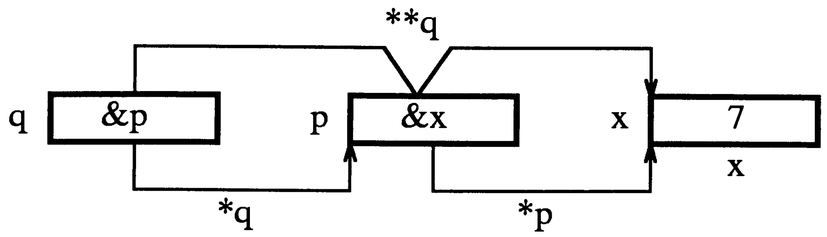
A csillagok száma természetesen növelhető. Megnyugtatásul annyit azonban meg kell jegyeznünk, hogy a szabványos C nyelv, a könyvtári függvényeket is figyelembe véve, maximálisan kétszeres indirektségű mutatókat használ.

A következő példaprogram kettős láncolást valósít meg a fenti három változó felhasználásával:

```
#include <stdio.h>

main()
{
 int x;
 int *p;
 int **q;

 x=7;
 p=&x;
 q=&p;
 x=x+ *p + **q;
 printf("x=%d\n",x);
}
```

Ránézésre meg tudjuk-e mondani, hogy mit ír ki a program ? Mielőtt ezt a kérdést megválaszolnánk, vessünk egy pillantást a 3.6. ábrára. Bonyolult

pointeres kapcsolatok értelmezésében hasznos segítséget jelent a grafikus ábrázolás.



3 6 ábra
Egyszeres és kétszeres indirektségű mutatók használata

Már említettük, hogy a C nyelven megadott típusok elég összetettek is lehetnek. A bonyolultság feloldásában, a deklarációk értelmezésében a mutatók esetén is ajánlott a **typedef** használata. (Emlékezzünk vissza, hogy kezdőérték nélküli változódeklaráció elé a **typedef**-et helyezve, a megadott típussal ekvivalens, de új névvel ellátott típushoz jutunk.) Az előző példa definíciói helyett használhatók az alábbi definíciók is:

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

vagy

```
typedef int *  iptr;    /*egészre mutató pointer típusa */
typedef iptr * ipptr;   /*iptr típusú objektumra mutató
                          pointer típusa*/
iptr p;
ipptr q;
```

A definíciók és a 3.6. ábra tanulmányozása után kijelenthetjük, hogy a példaprogram

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

utasítása az *x* változó értékét összegzi, méghozzá háromszor, vagyis a program által megjelenített szám, 21 lesz.

Ellenőrző kérdések

1. Ismertesse a definíció és a deklaráció fogalmak közötti különbséget!
2. Milyen szempontok szerint csoportosíthatjuk a C nyelv típusait?
3. Mit módosítanak a **long** és a **short** típusmódosítók?
4. Hogyan lehet előjeles és előjel nélküli egészeket deklarálni?
5. Hogyan lehet konstans értékű objektumot deklarálni?
6. Melyik a C nyelv "legkisebb" típusa?
7. Milyen összefüggés áll fenn a különböző hosszúságú egész típusok között?
8. Melyek a C nyelv lebegőpontos típusai?
9. Mi a változók szerepe a C nyelvben - hogyan kell saját változót létrehozni?
10. Ismertesse a konstansok használatának lehetőségeit!
11. Értelmezze az alábbi definíciókat:

```
int a, b = 12;
double pi=3.1416;
char ch;
unsigned long int cim = 0xb8000000L;
```

12. Mit értünk a balérték és a jobbérték fogalmán?
13. Milyen céllal használunk mutatókat?
14. Mennyi lesz az *a* változó értéke az alábbi programrészlet végrehajtása után?

```
main()
{
    int a = 7;
    int * p = &a;
   *p = a + *p - 2;
}
```

3.4. Operátorok és kifejezések

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

A operandusok a C nyelv azon elemei, amelyeken az operátorok fejtik ki hatásukat. Operandus lehet konstans érték, azonosító, sztring, függvényhívás, tömbindex kifejezés, tagkiválasztó kifejezés és tetszőleges összetett kifejezés, amely zárójelezett vagy zárójel nélküli, operátorokkal összekapcsolt további operandusokból áll. Ezeket az operandusokat elsődleges (*primary*) kifejezéseknek hívjuk.

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

```
op operandus        vagy        operandus op
```

Az első esetben, amikor az operátor (op) megelőzi az operandust előrevetett (prefixes), míg a második esetben hátravetett (postfixes) alakról beszélünk. Nézzünk néhány olyan kifejezést, ahol egyoperandusú operátorok szerepelnek:

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

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

```
operandus1 op operandus2
```


Ebben a csoportban a hagyományos aritmetikai műveletek mellett megtalálhatók a bitműveletek elvégzésére szolgáló operátorok is. Az alábbi példákban kétoperandusú operátorokat használtunk a kifejezésekben:

```
a+b        a!=b        a<<2        a+=b        a&0xff00
```

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

```
a < 0 ? -a : a
```

A 3.7. táblázatban a C nyelv összes operátorát felsoroltuk, röviden ismertetve az egyes műveletek működését.

Operátor	**Példa**	**Eredmény**
()	(e)	e (zárójelezett kifejezés)
	fv(a,b)	az fv függvény értéke
[]	t[2]	hivatkozás a 2 indexű tömbelemre
.	s.tag	hivatkozás a struktúra adattagjára
->	ps->tag	hivatkozás mutatóval kijelölt struktúra adattagjára
- (*unary*)	-a	az a negáltja
* (*unary*)	*p	hivatkozás a p által mutatott objektumra
& (*unary*)	&a	az a változó címe
~	~a	az a bitenkénti negáltja
++ (*prefix*)	++a	az a értéke a növelés után
++ (*postfix*)	a++	az a értéke a növelés előtt
-- (*prefix*)	--a	az a értéke a csökkentés után
-- (*postfix*)	a--	az a értéke a csökkentés előtt
sizeof	**sizeof**(t1)	a t1 típus mérete (byte-ban)
	sizeof e	az e kifejezés mérete (byte-ban)
(típus)	(t1)e	a t1 típusúvá konvertált e kifejezés
+	a + b	a és b összege
- (*binary*)	a - b	a és b különbsége
* (*binary*)	a * b	a és b szorzata
/	a / b	a és b hányadosa
%	a % b	az a/b művelet maradéka

3.7 táblázat
A C nyelv operátorainak összefoglaló táblázata

Operátor	Példa	Eredmény
<	a < b	1, ha a<b, 0 különben
>	a > b	1, ha a>b, 0 különben
<=	a <= b	1, ha a≤b, 0 különben
>=	a >= b	1, ha a≥b, 0 különben
==	a == b	1, ha a egyenlő b-vel, 0 különben
!=	a != b	1, ha a nem egyenlő b-vel, 0 különben
<<	a << b	az a bitjeinek balra tolása b bittel
>>	a >> b	az a bitjeinek jobbra tolása b bittel
& (*binary*)	a & b	a és b bitjei közötti ÉS kapcsolat
|	a | b	a és b bitjei közötti VAGY kapcsolat
^	a ^ b	a és b bitjei közötti kizáró VAGY kapcsolat
&&	a && b	a és b közötti logikai ÉS kapcsolat
||	a || b	a és b közötti logikai VAGY kapcsolat
!	!a	a logikai tagadása
=	a = b	a (a felveszi a b értéket)
+=	a += b	a + b (az eredmény a-ba kerül)
-=	a -= b	a - b (az eredmény a-ba kerül)
*=	a *= b	a * b (az eredmény a-ba kerül)
/=	a /= b	a / b (az eredmény a-ba kerül)
%=	a %= b	a % b (az eredmény a-ba kerül)
<<=	a <<= b	a << b (az eredmény a-ba kerül)
>>=	a >>= b	a >> b (az eredmény a-ba kerül)
&=	a &= b	a & b (az eredmény a-ba kerül)
|=	a |= b	a | b (az eredmény a-ba kerül)
^=	a ^= b	a ^ b (az eredmény a-ba kerül)
?:	a?e1:e2	az e1 kifejezés, ha a≠0, és az e2 kifejezés, ha a nulla.
,	e1, e2	e2 (e1 értékelődik ki először)
#		Előfordító:sztringbe illesztés
##		Előfordító:szövegbe illesztés
defined		Előfordító:makro létezésének vizsgálata

3 7 táblázat (folytatás)
A C nyelv operátorainak összefoglaló táblázata

Az operátorok részletes tárgyalása során a felhasználási lehetőségeik alapján csoportosítjuk az operátorokat. Mielőtt rátérnénk az operátorok ismertetésére,

ismerkedjünk meg néhány fontos fogalommal, amelyek a kifejezések kiértékelése során merülnek fel.

3.4.1. Precedencia és asszociativitás

Annak érdekében, hogy bonyolultabb kifejezéseket is helyesen tudjunk használni, meg kell ismerkednünk az elsőbbségi (precedencia) szabályokkal, amelyek meghatározzák a kifejezésekben szereplő műveletek kiértékelési sorrendjét. Az egyes operátorok közötti elsőbbségi kapcsolatot a 3.8. táblázatban foglaltuk össze. A táblázat sorai az azonos precedenciával rendelkező operátorokat tartalmazzák. A sorok végén külön jeleztük az azonos precedenciájú operátorokat tartalmazó kifejezésben a kiértékelés irányát, amit asszociativitásnak hívunk. A táblázat első sora tartalmazza a legnagyobb precedenciával rendelkező műveleteket.

Operátor	*Asszociativitás*
() [] . ->	balról jobbra
! ~ - ++ -- & * (típus) sizeof	jobbról balra
* / %	balról jobbra
+ -	balról jobbra
<< >>	balról jobbra
< <= > >=	balról jobbra
== !=	balról jobbra
&	balról jobbra
^	balról jobbra
|	balról jobbra
&&	balról jobbra
||	balról jobbra
?:	jobbról balra
= += -= *= /= %= <<= >>= &= |= ^=	jobbról balra
,	balról jobbra

3 8 táblázat
A C operátorok precedenciája és asszociativitása

A kifejezések kiértékelésénél két elsőbbségi szabályt vesz figyelembe a fordítóprogram. Nézzük meg részletesen ezeket az elsőbbségi szabályokat.

A precedencia szabály:

Az operátorok precedenciája akkor játszik szerepet a kifejezés kiértékelése során, ha a kifejezésben különböző precedenciájú műveletek szerepelnek. Ekkor mindig a magasabb precedenciával rendelkező operátort tartalmazó részkifejezés kerül először kiértékelésre, amit az alacsonyabb precedenciájú műveletek végrehajtása követ.

A műveletek kiértékelési sorrendjét úgy alakították ki, hogy az megfeleljen a szokásos matematikai műveleteknek. A kiértékelés sorrendje a matematikából ismert zárójelek segítségével meg is változtatható. (Felhívjuk a figyelmet arra, hogy C-ben csak a kerek zárójel () használható erre a célra.) Ennek megfelelően például az

```
5 + 3 * 4
```

kifejezés értéke 17 lesz. Ha az összeget zárójelbe tesszük, megváltoztatva ezzel a kiértékelés sorrendjét

```
(5 + 3) * 4
```

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

```
5 + (3 * 4)
```

akkor ismét 17-et kapunk eredményként, hisz ebben az esetben a zárójelezés csak kiemelte, de nem változtatta meg a számítás menetét.

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

Kifejezés:	**Ekvivalens kifejezés:**
`a & b || c`	`(a & b) || c`
`a = b || c`	`a = (b || c )`
`p==0 ? p+=1 : p+=2`	`(p==0 ? p+=1 : p) +=2`

Az első két kifejezés kiértékelhető, azonban az utolsó kifejezés esetén (bármelyik alakot megadva) hibajelzést kapunk (*Lvalue required*). Itt

mindenképpen zárójeleznünk kell ahhoz, hogy a kifejezés helyes és kiszámítható legyen:

```
(p==0) ? (p+=1) : (p+=2);
```

Az asszociativitás szabály:

Valamely kifejezésben több azonos precedenciájú művelet is szerepelhet. Ebben az esetben a 3.8. táblázat sorainak végén található útmutatást kell figyelembe venni a kiértékelés során, hisz a precedencia szabály nem alkalmazható. Az asszociativitás azt mondja meg, hogy az adott precedenciaszinten található műveleteket balról-jobbra vagy jobbról-balra haladva kell elvégezni.

Az értékadó utasítások csoportjában a kiértékelést jobbról-balra haladva kell elvégezni. Emiatt C-ben adott a lehetőség több változó együttes inicializálására:

```
int a,b,c;
a = b = c = 26;
```

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

```
a = (b =(c = 26));
```

3.4.2. Mellékhatások és a rövidzár kiértékelés

Bizonyos műveletek - a függvényhívás, a többszörös értékadás és a léptetés (növelés, csökkentés) - feldolgozása során a kifejezés értékének megjelenése mellett bizonyos változók is megváltozhatnak. Ezt a jelenséget mellékhatásnak (*side effect*) hívjuk.

A mellékhatások kiértékelésének sorrendjét nem határozza meg a C szabvány. Ezért javasolt minden olyan megoldás elkerülése, ahol a kifejezés eredménye függ a mellékhatások kiértékelésének sorrendjétől. A következő példában a különböző fordítók eltérően működő kódot hozhatnak létre. A ***pow*** függvény értéke függ attól, hogy az *n* léptetése megtörténik-e a ***pow*** meghívása előtt, vagy sem. (A TC 2.0-ban a léptetést a ***pow*** hívása után végzi el a fordító.)

```
#include <stdio.h>
#include <math.h>

main()
{
 int n=3;
 printf("%d %lf\n",++n, pow(2,n));

}
```

A program futásának eredménye (TC 2.0):

```
4 8.000000
```

Ilyen esetekben érdemes a függvényhívás kifejezést több részre bontani, a mellékhatást okozó részkifejezés kiértékelésének egyértelművé tételével:

```
 ++n;
 printf("%d %lf\n",n, pow(2,n));
```

Ezzel a futás eredménye minden C fordítónál ugyanaz lesz:

```
4 16.000000
```

A következő kifejezés kiértékelése szintén nem egyértelmű. A kérdés az, hogy az indexelést az *i* régi vagy új értékével végzi el a fordító.

```
a[i] = i++;
```

A C szabvány szándékosan nem rögzíti le a fenti és hasonló esetek értelmezését. Amikor ugyanis valamilyen mellékhatás (értékadás változónak) található kifejezésen belül, akkor a fordítóra van bízva a mellékhatások optimális feldolgozása, ami erősen függ a számítógép architektúrájától. Általában rossz gyakorlat az, ha a program kódja függ a kiértékelés sorrendjétől. Természetesen ahhoz, hogy elkerüljük ezeket a hibákat, ismernünk kell őket.

A 3.8. táblázatból látható, hogy a logikai kifejezések kiértékelése szintén balról-jobbra haladva történik. Bizonyos műveleteknél nem szükséges a teljes kifejezést kiértékelni ahhoz, hogy egyértelmű legyen a kifejezés értéke.

Példaként vegyük a logikai ÉS (&&) operátort, amely használata esetén a baloldali operandus 0 értéke esetén a jobboldali operandus kiértékelése

feleslegessé válik. Ezt a kiértékelési módot rövidzár (*short-circuit*) kiértékelésnek nevezzük.

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

```
x || y++
```

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

3.4.3. Elsődleges operátorok

Az első csoportba azok az operátorok tartoznak (`()`, `[]`, `->`, `.`), amelyekkel a későbbiek során ismerkedünk meg részletesen, hiszen hozzájuk speciális C nyelvi szerkezetek kapcsolódnak. Nevezetesen a függvényhívás (`fv(argumentumok)`), a tömb indexelés (`tomb[index]`) és a struktúra tagra való hivatkozás (`str.tag` vagy `pstr->tag`) operátorokról van szó.

A zárójel () operátor azonban kettős szereppel rendelkezik. Mint már említettük, zárójelek segítségével a kifejezések kiértékelése megváltoztatható. A zárójelbe helyezett kifejezés típusa és értéke megegyezik a zárójel nélküli kifejezés típusával és értékével.

3.4.4. Aritmetikai operátorok

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

`26 / 4`	a kifejezés értéke (a hányados)	`6`
`26 % 4`	az kifejezés értéke (a maradék)	`2`

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

```
(a / b) * b + (a % b) = a
```

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

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

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

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

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

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

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

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

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

3.4.5. Összehasonlító és logikai operátorok

A C nyelv nem rendelkezik külön logikai adattípussal, azonban vannak olyan utasításai, melyekben feltételeket kell definiálnunk (**if**, **while**, stb). A feltételek tetszőleges kifejezések lehetnek, melyek nulla vagy nem nulla értéke szolgáltatja a logikai hamis vagy igaz eredményt.

A feltételekben gyakran kell összehasonlítanunk bizonyos értékeket, hogy a program további működéséről döntsünk. Az összehasonlítás elvégzésére az összehasonlító operátorokat használjuk, melyeket az alábbi táblázatban foglaltuk össze:

Matematikai alak	C kifejezés	Jelentés
`a < b`	`a < b`	a kisebb mint b
`a ≤ b`	`a <= b`	a kisebb vagy egyenlő mint b
`a > b`	`a > b`	a nagyobb mint b
`a ≥ b`	`a >= b`	a nagyobb vagy egyenlő mint b
`a = b`	`a == b`	a egyenlő b-vel
`a ≠ b`	`a != b`	a nem egyenlő b-vel

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

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

```
x >= 0
```

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

```
-5 < x < 5
```

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

Ahhoz, hogy bonyolultabb feltételeket is össze tudjunk hasonlítani, a relációs operátorok mellett szükségünk van a logikai operátorokra is. C-ben a logikai ÉS (&&) és a logikai VAGY (||) műveletek használhatók a feltételek megfogalmazása során. Az előző feltétel egyszerűen felírható a logikai ÉS műveletet használva:

```
-5 < x && x < 5
```

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

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

```
!(-5 >= x || x >= 5)
```

(A logikai tagadás során minden relációt az ellentétes irányú relációra, az ÉS operátort pedig a VAGY operátorra (illetve fordítva) kell cserélni.)

A C programokban gyakran használjuk az

```
ok == 0
```

kifejezés helyett a

```
!ok
```

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

A logikai operátorok működését ún. igazságtáblával írják le. Az alábbiakban felrajzoltuk mindhárom logikai művelet igazságtábláját:

a	!a
0	1
1	0

logikai tagadás

a	b	a&&b
0	0	0
0	1	0
1	0	0
1	1	1

logikai ÉS művelet

a	b	a||b
0	0	0
0	1	1
1	0	1
1	1	1

logika VAGY művelet

3 9 ábra
A logikai operátorok igazságtáblája

3.4.6. Léptető operátorok

A változók értékét léptető operátorok a magas szintű nyelvekben csak ritkán fordulnak elő. C nyelv lehetőséget biztosít valamely változó értékének eggyel való növelésére ++ (*increment*), illetve eggyel való csökkentésére -- (*decrement*).

A léptető operátorok az aritmetikai típusokon kívül a mutatókra is alkalmazhatók, ahol azonban nem 1 byte-tal való elmozdulást, hanem a szomszédos elemre való léptetést jelentik.

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

```
main()
{
     int a;

     /* prefixes alak: */
     ++a;                    --a;

     /* postfixes alak: */
     a++;                    a--;
}
```

Ha az operátorokat a fenti programban bemutatott módon használjuk, nem látszik különbség az előrevetett és hátravetett forma között, hisz mindkét esetben a változó értéke léptetődik.

Ha azonban az operátort bonyolultabb kifejezésben alkalmazzuk, akkor a prefixes alak használata esetén a léptetés a kifejezés kiértékelése előtt megy végbe és a változó az új értékével vesz részt a kifejezés kiértékelésében:

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

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

A hátravetett alakot használva a kifejezés kiértékelésében a változó az eredeti értékével vesz részt és a léptetés a kiértékelés után kerül végrehajtásra:

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

A kifejezés feldolgozása után az *x* változó értéke *5 0*, míg az *n* változó értéke *6 0* lesz.

Az eggyel való növelés és csökkentés hagyományos megadása

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

helyett mindig érdemes a megfelelő léptető operátort használni,

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

mely az áttekinthetőség mellett, gyorsabb kód létrehozását is eredményezi.

A C nyelvre jellemző, hogy valamely operátor-karakter, több értelmezéssel is használható. Az adott művelet (pl. -) értelmezése attól a szövegkörnyezettől függ, amelyben használjuk. Az alábbi példában a + és a - jeleket különböző műveletek kijelölésére használtuk. Az egyes kifejezések kiértékelése utáni állapotot megjegyzésben tüntettük fel (LEPTET.C):


```
#include <stdio.h>

main()
{
     int i,j,k;

     i=3; j=1; k=2;                    /*  i  j  k  */
     k=i+++j;
     printf("%d %d %d\n",i,j,k);       /*  4  1  4  */

     i=3; j=1; k=2;
     k= ++i+j++;
     printf("%d %d %d\n",i,j,k);       /*  4  2  5  */

     i=3; j=1; k=2;
     k= --i-j--;
     printf("%d %d %d\n",i,j,k);       /*  2  0  1  */

     i=3; j=1; k=2;
     k= i---j;
     printf("%d %d %d\n",i,j,k);       /*  2  1  2  */

     i=3; j=1; k=2;
     k= -i+++j;
     printf("%d %d %d\n",i,j,k);       /*  4  1 -2  */

     i=3; j=1; k=2;
     k= ++i-j++;
     printf("%d %d %d\n",i,j,k);       /*  4  2  3  */

     i=3; j=1; k=2;
     k= +i+-j;
     printf("%d %d %d\n",i,j,k);       /*  3  1  2  */

     i=3; j=1; k=2;
     k= -i--+j;
     printf("%d %d %d\n",i,j,k);       /*  2  1 -2  */

     i=3; j=1; k=2;
     k= i--+--j;
     printf("%d %d %d\n",i,j,k);       /*  2  0  3  */

     i=3; j=1; k=2;
     k+=++i+--j;
     printf("%d %d %d\n",i,j,k);       /*  4  0  6  */

     i=3; j=1; k=2;
     k+=-i+++j;
     printf("%d %d %d\n",i,j,k);       /*  4  1  0  */
}
```

A példaprogramban az egyes operátorok felderítését a fordítóra bíztuk, amely a szövegkörnyezetnek és a prioritásoknak megfelelően értelmezi a műveleteket. Ez a programírási gyakorlat nem helyes, hiszen szóközök elhelyezésével a programozó számára is érthetővé lehet tenni a kijelölt műveleteket. Az utolsó két sorban szereplő kifejezésekben összetett értékadások szerepelnek.

3.4.7. Bitműveletek

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

3.4.7.1. Bitenkénti logikai műveletek

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

Operátor	Művelet
~	1-es komplemens
&	bitenkénti ÉS
|	bitenkénti VAGY
^	bitenkénti kizáró VAGY

A bitenkénti logikai műveletek működésének leírását a 3.10 ábra tartalmazza, ahol a 0 és az 1 számjegyek a törölt, illetve a beállított bitállapotot jelölik.

a	b	a & b	a | b	a ^ b	~a
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

3.10 ábra
A bitenkénti logikai operátorok igazságtáblája

A bitenkénti logikai műveletek a C nyelv szintjén biztosítják a számítógép hardver elemeinek programozását. A perifériák többségének alacsonyszintű vezérlése bizonyos bitek beállítását, illetve törlését jelenti. Ezeket a műveleteket összefoglaló néven "maszkolásnak" nevezzük.

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

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

A fenti szó értéke hexadecimális számrendszerben 0x07CA, illetve decimális alakban 1994.

Bitek 1-be állítása

A maszk azokban a bitpozíciókban, ahol a beállítást el kell végezni, 1-et, míg máshol 0-át tartalmaz. A beállítást a maszk elkészítése után a VAGY művelettel végezzük el. Mivel C-ben nem lehet kettes számrendszerben megadni konstansokat, a 16-os számrendszert használjuk a maszk megadásához.

Állítsuk az *ax* változó 5. és 12. bitjét 1-be! A megoldást az alábbi programmal végezhetjük el. (A ***printf*** hívásnál használt %X formátummal hexadecimális alakban jelenítjük meg az eredményt.)

```
#include <stdio.h>
main()
{
     int ax=1994;
     printf("ax= %X\n", ax);     /* 0x07CA = 1994 */
     ax=ax | 0x1020;
     printf("ax= %X\n", ax);     /* 0x17EA = 6122 */
}
```

Ellenőrizzük bitek szintjén a művelet elvégzését:

```
  0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax = 0x07CA
| 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0      maszk = 0x1020
-----------------------------------
  0 0 0 1 0 1 1 1 1 1 1 0 1 0 1 0        ax = 0x17EA
```

Bitek törlése

Bitek törlése esetén a bitmaszk az előző műveletnél használt maszk ellentettje. Azokba a pozíciókba, ahol a törlendő bitek állnak 0-át teszünk, míg a maszk további bitjei egyesek lesznek. A maszkot ezek után bitenkénti ÉS műveletben kell szerepeltetni.

Az alábbi példában az *ax* változó értékének 10. és 3. bitjeit töröljük. A maszk felírásánál egyaránt használhatjuk a `0xFBF7` vagy a `~0x0408` alakot.

```
#include <stdio.h>
main()
{
     int ax=1994;
     printf("ax= %X\n", ax);     /* 0x07CA = 1994 */
     ax=ax & 0xFBF7;
     printf("ax= %X\n", ax);     /* 0x03C2 =  962 */
}
```

Ellenőrizzük bitek szintjén a művelet elvégzését:

```
  0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax = 0x07CA
& 1 1 1 1 1 0 1 1 1 1 1 1 0 1 1 1      maszk = 0xFBF7
-----------------------------------
  0 0 0 0 0 0 1 1 1 1 0 0 0 0 1 0        ax = 0x03C2
```

Ha bitek beállítását és törlését egyaránt el kell végeznünk, akkor a fenti két műveletet egymás után kell alkalmaznunk.

A kizáró VAGY művelet használata

A kizáró VAGY művelet érdekes tulajdonsága, hogy ugyanazt a maszkot egymás után kétszer használva visszakapjuk a kiindulási értéket. Így a kizáró VAGY műveletet jól felhasználható titkosításra, bitek ki-be kapcsolására, mint ahogy az a példából is látható:


```
#include <stdio.h>
main()
{
     int ax=1994;
     printf("ax= %X\n", ax);     /* 0x07CA =  1994 */
     ax=ax ^ 0x8421;
     printf("ax= %X\n", ax);     /* 0x83EB = -31765 */
     ax=ax ^ 0x8421;
     printf("ax= %X\n", ax);     /* 0x07CA =   1994 */
}
```

A műveletek bit szintű áttekintése:

```
  0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax = 0x07CA
^ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1      maszk = 0x8421
-----------------------------------
  1 0 0 0 0 0 1 1 1 1 1 1 0 1 1 1        ax = 0x83EB
^ 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1      maszk = 0x8421
-----------------------------------
  0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax = 0x07CA
```

Az alábbi példában a kizáró VAGY műveletet két egész típusú változó értékének felcserélésére használjuk. A cserét egy harmadik változó bevezetése nélkül végezzük.

```
#include <stdio.h>
main()
{
     int x=25000, y=20000;

     printf("\nA csere előtt:\n");
     printf("x = %d, y = %d\n", x, y);

     x = x^y;          /* A csere */
     y = y^x;
     x = x^y;

     printf("\nA csere után:\n");
     printf("x = %d, y = %d\n", x, y);
}
```

3.4.7.2. Biteltoló műveletek

A bitműveletek másik csoportjába, a biteltoló (*shift*) operátorok tartoznak. Az eltolás balra (<<) és jobbra (>>) egyaránt elvégezhető. Az eltolás során a baloldali operandus bitjei annyiszor lépnek balra (jobbra), amennyi a jobboldali operandus értéke. A felszabaduló bitpozíciókba 0-ás bitek kerülnek, míg a kilépő bitek elvesznek.

Induljunk ki újra az *1994*-es értékből, és program segítségével végezzük el az `1994 << 1` és az `1994 >> 2` műveleteket

```
#include <stdio.h>
main()
{    int ax;

     ax=1994;
     printf("ax= %X\n", ax);     /* 0x07CA =  1994 */
     ax=ax << 1;
     printf("ax= %X\n", ax);     /* 0x0F94 =  3988 */

     ax=1994;
     printf("ax= %X\n", ax);     /* 0x07CA =  1994 */
     ax=ax >> 2;
     printf("ax= %X\n", ax);     /* 0x01F2 =   498 */
}
```

Ha megnézzük az *ax* bitjeit, jól látható a művelet menete:

```
0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax
0 0 0 0 1 1 1 1 1 0 0 1 0 1 0 0        ax << 1

0 0 0 0 0 1 1 1 1 1 0 0 1 0 1 0        ax
0 0 0 0 0 0 0 1 1 1 1 1 0 0 1 0        ax >> 2
```

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

Előjeles egészek esetén a jobbra történő biteltolás előjeltartó, vagyis az eltolás után a legmagasabb helyiértékű előjel bit visszaíródik az eredeti helyére:

```
int ax = -2;
```

esetén:

```
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0        ax
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1        ax >> 1
```

Ha azonban előjel nélküli egészet használunk:

```
unsigned int ax = -2;

1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0        ax
0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1        ax >> 1
```

Ezt a különbséget szem előtt kell tartanunk bonyolultabb bitenkénti logikai és biteltoló műveleteket egyaránt tartalmazó kifejezések felírása során.

Írjunk egy olyan programot, amely a beolvasott egész számot két byte-ra bontja. A feladat aritmetikai és bitenkénti műveletek felhasználásával egyaránt megoldható. Nézzünk egy lehetséges megoldást, amelyben bitműveleteket használtunk.

```
#include <stdio.h>
main()
{
     int num;
     unsigned char lo, hi;

     /* A szám beolvasása */
     printf("\nKérek egy egész számot: ");
     scanf("%d",&num);
     printf("A szám %d  = %X\n", num, num);

     /* Az alsó byte meghatározása maszkolással */
     lo=num & 0x00FF;
     printf("A szám alsó byte-ja : %X\n", lo);

     /* Az felső byte meghatározása biteltolással */
     hi=num >> 8;
     printf("A szám felső byte-ja: %X\n", hi);
}
```

3.4.8. Értékadó operátorok

Már említettük, hogy C nyelvben az értékadás sajátos módon megy végbe. Egyrészről az értékadás egy olyan kifejezés, amely a baloldali operandus által kijelölt objektumnak adja a jobboldalon megadott kifejezés értékét, másrészt pedig ez az érték egyben az értékadó kifejezés értéke is. Ebből következik, hogy értékadás tetszőleges kifejezésben szerepelhet.

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

```
a = 13;
b = (a+4)*7-30;
```

során az *a* változó értéke *13*, míg a *b* változóé *89* lesz. Felírható azonban olyan, más nyelvektől idegen, kifejezés is,

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

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

Ha megnézzük a 3.8. táblázatot, láthatjuk, hogy az értékadó operátorok kiértékelése jobbról-balra haladva történik. Emiatt C nyelven használható a többszörös értékadás, melynek során több változó veszi fel ugyanazt az értéket:

```
a = b = 26;
```

A kiértékelés a `b = 26` értékadó kifejezés feldolgozásával kezdődik, melynek során a *b* változó felveszi a *26* értéket, ami egyben ezen részkifejezés értéke is. Ezt követi az `a = részkifejezés_értéke` értékadás értelmezése, melynek végeztével az *a* változó is felveszi a *26* értéket, és a teljes kifejezés értéke is *26* lesz.

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

```
a = a + 2;
```

Az ilyen alakú kifejezések tömörebb formában is felírhatók:

```
a += 2;
```


Általában elmondható, hogy a

$$\texttt{kif}_1 = \texttt{kif}_1 \ \texttt{op} \ \texttt{kif}_2$$

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

$$\texttt{kif}_1 \ \texttt{op} = \texttt{kif}_2$$

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

Hagyományos forma	*Tömör forma*
`a = a + b`	`a += b`
`a = a - b`	`a -= b`
`a = a * b`	`a *= b`
`a = a / b`	`a /= b`
`a = a % b`	`a %= b`
`a = a << b`	`a <<= b`
`a = a >> b`	`a >>= b`
`a = a & b`	`a &= b`
`a = a | b`	`a |= b`
`a = a ^ b`	`a ^= b`

Meg kell jegyeznünk, hogy az

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

kifejezésnek megfelelő hagyományos értékadás az

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

nem pedig az

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

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

3.4.9. Pointer operátorok

A C nyelvben található két olyan speciális egyoperandusú művelet, amelyeket mutatókkal kapcsolatban használunk. A "címe" (&) művelet eredménye az operandusként megadott memóriaobjektum címe:

```
int a, *ptr;

ptr = &a;
```

A "címe" operátort arra használjuk, hogy mutatóinkat már meglévő változókra irányítsuk.

A másik mutató operátor (*) az indirekt hivatkozás elvégzéséhez szükséges:

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

A `*ptr` kifejezés a *ptr* pointer által mutatott objektumot jelenti.

A mutatókkal a későbbiekben még részletesen foglalkozunk, most csak a mutatókkal használható, nem pointer operátorok áttekintésére szorítkozunk. Alapvetően az aritmetikai operátoroknál ismertetett pointer aritmetikai műveletekről van szó, amelyek más operátorokkal is kijelölhetők. A mutató léptetése a szomszédos elemre többféle módon is elvégezhető:

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

Az előző elemre való visszalépésre szintén több lehetőség közül választhatunk:

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

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

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


A léptető és az indirektség operátorok együttes használatához kellő óvatosság szükséges. (A példában az *sp* mutatóval egy konstans sztringre mutatunk.)

```
char *sp = "C-nyelv";
```

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

```
printf("%c\n",*sp++);
```

kifejezés kiértékelése után? A precedencia táblázatban azt találjuk, hogy az indirektség és a léptető operátorok azonos precedenciával rendelkeznek. A kérés megválaszolásánál az asszociativitási szabályt, a kiértékelés jobbról-balra történő menetét kell figyelembe vennünk. Ennek alapján először a léptetés operátorát értelmezi a rendszer, amely azonban csak a teljes kiértékelés után fejti ki hatását, hisz a hátravetett alakot használtuk. Ezt követi az indirektség operátorának feldolgozása, melynek hatására a 'C' betűt írja ki a ***printf*** függvény. A kiértékelés végén az *sp* tovább lép és a '-' karakterre mutat.

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

```
printf("\'%c\'\n",kifejezés);
printf("\"%s\"\n",sp);
```

programrészlet által kiírt karakter, illetve szöveg található:

Kifejezés	*Karakter*	*Szöveg*
`*++sp`	`'-'`	`"-nyelv"`
`*sp++`	`'C'`	`"-nyelv"`
`++*sp`	`'D'`	`"D-nyelv"`
`(*sp)++`	`'C'`	`"D-nyelv"`

3.4.10. A sizeof operátor

A C nyelv tartalmaz egy olyan fordítás idején kiértékelésre kerülő egyoperandusú operátort, amely megadja tetszőleges objektum méretét. A

```
sizeof objektum
```

vagy a

```
sizeof (típusnév)
```

alakú kifejezések értéke egy olyan egész szám, amely megegyezik a megadott objektum, illetve típus byte-ban kifejezett méretével. (Pontosabban a **sizeof** operátor a STDDEF.H állományban definiált ***size_t*** típusú előjel nélküli egész értéket szolgáltat.)

Az *objektum* tetszőleges egyszerű változó, tömb vagy struktúra egyaránt lehet:

```
int a;
double d;

a = sizeof d;       /* A d változó mérete */
```

A *típusnév* az alaptípusokon túlmenően tetszőleges származtatott típust is jelölhet:

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

3.4.11. A vessző operátor

Egyetlen kifejezésben több, akár egymástól független kifejezés is elhelyezhető, a vessző operátor felhasználásával. A vessző operátort tartalmazó kifejezés balról-jobbra haladva kerül kiértékelésre, és a kifejezés értéke és típusa megegyezik a jobboldali operandus értékével, illetve típusával.

Példaként tekintsük az

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


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

A vessző operátort gyakran használjuk különböző változók kezdőértékeinek egyetlen utasításban (kifejezésben) történő beállítására:

```
int x, y;
double z;

x = 5, y = 0, z = 1.2345 ;
```

(Ez a megoldás jól használható például a **for** ciklus szervezésénél.) Ugyancsak a vessző operátort kell használnunk, ha két változó értékét egyetlen utasításban kívánjuk felcserélni (harmadik változó felhasználásával):

```
int a=13, b=26, c;

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

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

3.4.12. A feltételes operátor

A feltételes operátor (?:) három operandussal rendelkezik:

```
kif1 ? kif2 : kif3
```

A feltételes kifejezésben először a `kif1` kifejezés kerül kiértékelésre. Amennyiben ennek értéke nem nulla (igaz), akkor a `kif2` értéke adja a feltételes kifejezés értékét. Ellenkező esetben a kettőspont után álló `kif3` értéke lesz a feltételes kifejezés értéke. Ily módon a kettőspont két oldalán álló kifejezések közül mindig csak az egyik értékelődik ki. A feltételes kifejezés típusa a nagyobb pontosságú részkifejezés típusával egyezik meg. Az

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

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

Megoldás az **if** utasítás felhasználásával:

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

Feltételes kifejezéssel sokkal tömörebben oldható meg a feladat:

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

Felhívjuk a figyelmet arra, hogy az **if** automatikus átírása:

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

fordítási hibához vezet, hiszen a precedencia szabályokat figyelembe véve - a ?: operátor precedenciája magasabb az értékadó operátorokénál - az alábbi zárójelezés adja a műveletek alapértelmezés szerinti csoportosítását:

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

A külső zárójelben található érték nem balérték, ezért lép fel hiba a fordítás folyamán. Az átírás azonban helyessé tehető zárójelek megfelelő elhelyezésével:

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

Nézzünk két jellegzetes példát a feltételes operátor alkalmazására. Az első esetben a *ch* karakter kiírásakor a felhasználandó formátumot feltételes kifejezéssel adjuk meg. Ha a *ch* karakter vezérlő karakter (kódja < 32), akkor a hexadecimális kódját, ellenkező esetben pedig magát a karaktert írjuk ki:

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

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

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


3.4.13. Típuskonverziók

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

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

A típuskonverziót azonban a programozó is előírhat a C programban, a típuskonverziós operátor (*cast*) felhasználásával. Ez az egyoperandusú operátor a konvertálandó kifejezés előtt zárójelek között tartalmazza a típusnevet:

```
(típusnév) kifejezés
```

Mivel ebben az esetben a típusnév megjelenik a konverziós előírásban, *explicit* típuskonverzióról beszélünk.

3.4.13.1. Implicit típuskonverziók

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

```
int i=5, j;          float f=3.65;
i + f;
```

Az implicit konverziók azonban nem minden esetben mennek végbe információvesztés nélkül. Az értékadás és a függvényhívás paraméterezése során tetszőleges típusok közötti konverzió is előfordulhat. Ha a fenti példában az összeget a *j* változónak feleltetjük meg

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

akkor bizony adatvesztés lép fel, hiszen az összeg törtrésze elvész, és *8* lesz a *j* változó értéke.

A következőkben áttekintjük a különböző típusú objektumok közötti konverziók végrehajtásának szabályait.

*Felsorolási (**enum**) konstansok konverziója*

Ezek a konstansok mindig **int** típusúra konvertálódnak.

Lebegőpontos típusok konverziója

A lebegőpontos típus egész jellegű típussá való átalakítása, a törtrész eldobásával megy végbe. Ha azonban az eredmény nem fér el a megadott egész típusban, akkor annak értéke határozatlan lesz.

Amikor a **float** típus **double** vagy **long double** típusra, illetve a **double** típus **long double** típusra konvertálódik az objektum értéke változatlan marad.

Valamely lebegőpontos típus kisebb lebegőpontos típussá is konvertálható. Amennyiben a konvertált érték nem ábrázolható az új típussal, az eredmény határozatlan lesz. Ha az érték csak kisebb pontossággal ábrázolható a megadott típussal, akkor a konverzió kerekítéssel megy végbe.

Egész jellegű típusok konverziója

A **char**, a **short int**, a felsorolt típus és a bitmezők minden olyan esetben alkalmazhatók, ahol az **int** típus használható. Ezek a típusok automatikusan **int** típussá konvertálódnak. Ha azonban az **int** típus nem alkalmas az értékük tárolására, akkor **unsigned int** lesz a konverzió céltípusa. Ez a konverziós szabály az "egész konverzió" (*integral promotion*) nevet viseli. Mivel a fenti konverzió érték- és előjelhelyes eredményt ad, értékmegőrző konverziónak is szokás nevezni.

Az egész típus mindig átalakítható lebegőpontos típussá. Amennyiben a lebegőpontos típus pontossága nem elegendő az egész érték pontos tárolására, akkor kerekítés történik.

Előjeles és előjel nélküli egészek konverziója

Az alábbi szabályok akkor érvényesülnek, amikor előjeles és előjel nélküli egész típusok konvertálódnak más egész jellegű típussá.

- Ha a pozitív, *signed* egész típus konverziójának céltípusa egy ugyanolyan vagy nagyobb előjel nélküli egész típus, akkor az érték változatlan marad.
- Ha egy negatív egész számot ugyanolyan vagy nagyobb méretű előjel nélküli egésszé kívánunk átalakítani, akkor a konverzió első lépésében a céltípus előjeles megfelelője keletkezik. Majd ezt az előjeles értéket eggyel megnövelve, az adott hosszúságon ábrázolható maximális *unsigned* értékhez hozzáadva keletkezik a konverzió eredménye. (2-es komplemens ábrázolás esetén a kiindulási bitminta megmarad, és az üres helyek az előjelbittel töltődnek fel.)
- Amikor egy előjeles vagy előjel nélküli egészet kisebb *unsigned* egésszé alakítunk át, annak értéke egy olyan nem negatív maradék lesz, amit úgy kapunk, hogy a kiindulási értéket elosztjuk a céltípus lehetséges legnagyobb értéke + eggyel.
- Ha az előjeles vagy előjel nélküli egészet kisebb *signed* egésszé kívánjuk konvertálni, de az nem ábrázolható a céltípussal, akkor a konverzió eredménye implementációfüggő.
- Amikor egy előjel nélküli egész kovertálódik ugyanolyan méretű *signed* egésszé, de annak értéke nem ábrázolható a céltípussal, akkor a konverzió eredménye implementációfüggő.

A szokásos aritmetikai konverziók

A kétoperandusú műveletek többsége az operandusait és az eredményt a "legnagyobb" közös típusúra konvertálja. Az átalakítás során felhasznált szabályok a "szokásos aritmetikai konverziók" nevet viselik. Az alábbiakban bemutatott konverziós szabályok a legnagyobb pontosságú típustól az **int** típusig tartanak.

- Ha valamelyik operandus **long double** típusú, akkor a másik operandus is **long double** típusúvá konvertálódik.
- Ha valamelyik operandus **double** típusú, akkor a másik operandus is **double** típusúvá konvertálódik.
- Ha valamelyik operandus **float** típusú, akkor a másik operandus is **float** típusúvá konvertálódik.

- Ha valamelyik operandus **unsigned long int** típusú, akkor a másik operandus is **unsigned long int** típusúvá konvertálódik.
- Ha az egyik operandus **long int** típusú és a másik operandus típusa **unsigned int**, akkor a másik operandus csak akkor konvertálódik **long int** típusúvá, ha az képes az **unsigned int** érték tárolására. Ellenkező esetben minkét operandus **unsigned long int** típusú lesz.
- Ha valamelyik operandus **long int** típusú, akkor a másik operandus is **long int** típusúvá konvertálódik.
- Ha valamelyik operandus **unsigned int** típusú, akkor a másik operandus is **unsigned int** típusúvá konvertálódik.
- Amennyiben a fenti szabályok egyike sem került alkalmazásra, akkor mindkét operandus **int** típusú. (Az egész konverzió következtében.)

Mutató (pointer) konverziók

Egy adott típusra mutató pointer tetszőleges más típusra mutató pointerré konvertálható. Mivel azonban az ilyen típus-átalakítások hibalehetőséget hordozanak magukban a fordítóprogramok figyelmeztetnek az ilyen jellegű implicit konverziókra. A *0* (nulla) értékű kifejezések és a **void** típusú adatra mutató pointerek (`void *`) figyelmeztetés nélkül konvertálódnak tetszőleges más típusú mutatóvá.

A mutatók *implicit* módon előjel nélküli egész értékekké is átalakíthatók, amennyiben az egész típus elegendően nagy a pointer értékének tárolására. Az egész típusú kifejezések mutatókká alakíthatók, azonban ehhez *explicit* típuskonverziót kell használnunk. Mindkét konverzió implementációfüggő.

3.4.13.2. Az explicit típuskonverzió

A implicit típuskonverzió menete a felhasználói típuselőírás (*type cast*) operátorának segítségével módosítható. A

```
(típusnév) kifejezés
```

szerkezetben a *kifejezés* a *típusnévben* megadott típusúvá konvertálódik, az előző alfejezetben ismertetett konverziós szabályok alapján.

Az explicit típuskonverziót gyakran használjuk, a könyvtári függvények helyes (megfelelő típusú argumentummal történő) meghívásához:

```
int a;
a = sqrt( (double) 25);
```

implicit **double-int** konverzió

explicit **int-double** konverzió

A mutatókkal kapcsolatosan gyakran használjuk a típuselőírás operátort. Az alábbi példában az *iptr* által kijelölt **int** adatot byte-onként is elérhetjük a *cprt* mutató segítségével::

```
char * cprt;
int * iptr;
. . .
cprt = (char *) iptr;
```

A szabványtól eltérően a legtöbb IBM PC C fordító megengedi, hogy a típuselőírás balértékként szerepeljen a kifejezésekben, így értelmezettek az alábbi kifejezések:

```
* (char far *) 0xB8000000L = 'X';

char *p;
. . .
++(long *)p;  /* 4 byte-tal lép a p mutató */
```

A következő példában a **float** aritmetikával elvégzett számítás helyett pontosabb **double** műveletvégzést írunk elő:

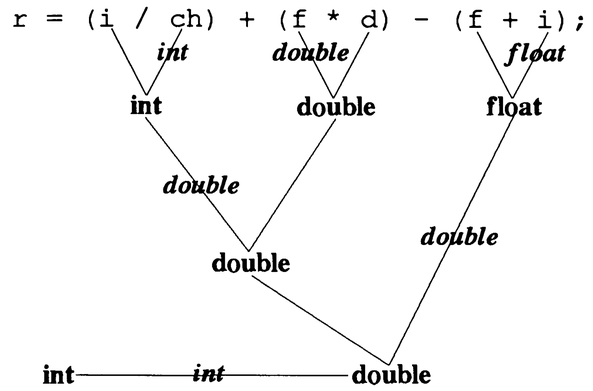
```
float f1, f2, f3;
. . .
f3 = (double) f1 * f2;
```

Az utolsó példánkban egy bonyolultabb kifejezés kiértékelése során elvégzett implicit konverziókat ábrázoljuk. Ezt követően explicit típuskonverziókkal úgy avatkozunk be a kiértékelés menetébe, hogy a kifejezés értéke lényegesen ne változzék.

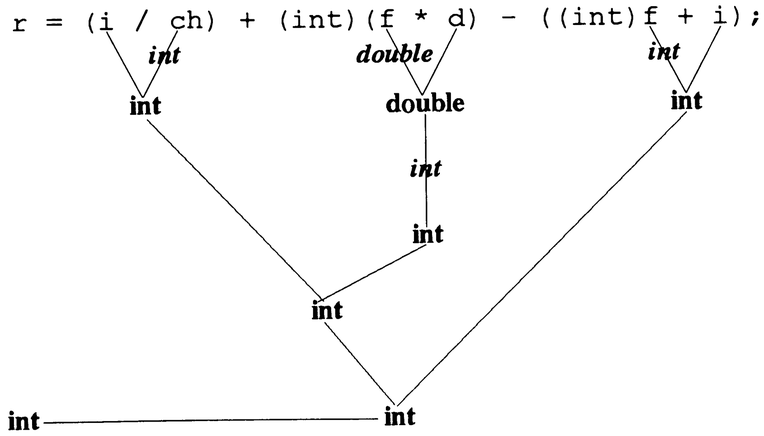
```
#include <stdio.h>
main()
{
 char   ch = 34;
 int    i  = 123;
 float  f  = 3.1234e2;
 double d  = 456.123e-1;
 int    r;

 r = (i / ch) + (f*d) - (f+i);
 printf("%ld\n",r);

 r = (i / ch) + (int)(f*d) - ((int)f+i);
 printf("%ld\n",r);
}
```

A program mindkét esetben ugyanazt az eredmény adja (*13814*). Mivel a két kifejezés értéke függ a változók kezdőértékétől, az ehhez hasonló megoldásoknál mindig körültekintően kell eljárnunk.

A 3.11. és 3.12. ábrákon a példában szereplő kifejezések kiértékelése során elvégzett konverziókat dőlt, míg a részeredmények típusát vastagított típusnevek jelölik.



3 11 ábra
A kiértékelés menete implicit konverziókkal



3 12 ábra
A kiértékelés menete explicit és implicit konverziókkal

Láthatjuk, hogy az első esetben hat implicit konverziót hajt végre a fordító, míg a második kifejezésben csak négy (explicit és implicit) konverziót kell elvégeznie. Ugyancsak jól látható az ábrákon, hogy az első kifejezés kiszámításánál többségében **double** lebegőpontos műveletek kerülnek végrehajtásra, míg a második kifejezés kiértékelése alapvetően egész aritmetikával történik.

A kifejezések egyszeri kiértékelése során, nem érzékelhető a kiértékelések közötti időkülönbség Ha azonban a kifejezést több ezerszer kell kiértékelni, például ciklusban, akkor már jelentős az első kifejezés kiszámításához elhasznált időtöbblet.

Ellenőrző kérdések

1 Csoportosítsa a C nyelv műveleteit az operandusok száma szerint!
2. Ismertesse az operátor, az operandus és a kifejezés fogalmak kapcsolatát!
3 Mondjon példát előrevetett és postfix operátorokra!
4 Milyen szabályok érvényesülnek a kifejezések kiértékelésénél?
5 Mit mond ki a precedencia szabály?
6 Mikor alkalmazza fordító az asszociativitás szabályt?
7 Csoportosítsa az aritmetikai operátorokat!
8 Csoportosítsa a logikai és relációs operátorokat!

9. Mire használjuk a léptető operátorokat?
10. Ismertesse a C nyelv bitműveleteit!
11. Mi teszi lehetővé a többszörös értékadás használatát?
12. Csoportosítsa az aritmetikai operátorokat!
13. Milyen műveletek végezhetők mutató típusú változókon?
14. Hogyan kérdezhető le egy C objektum mérete?
15. Mi lesz az *a*, a *b* és a *c* változók értéke az egyes kifejezések kiértékelése után?

```
int a , b , c;
a = b =10;
c = a++ * (b % 4);
c =  a>b ? a+2 : b-3;
b--, a++;
c = sizeof(a) + sizeof( double);
```

16. Miért van szükség a típusok konverziójára?
17. Milyen konverziókat hajt végre a rendszer az alábbi kifejezés kiértékelésénél?

```
int a = 10;
double d =3.14;
char ch = 54;
long r;
r = (int)(d + a*ch);
```

18. Mikor "igaz" és mikor "hamis" egy C logikai kifejezés?
19. Írjon fel egy olyan kifejezést, amely akkor ad igaz értéket, ha az *x* változó értéke 10-nél nagyobb, de 3-mal és 5-tel sem osztható.
20. Írja az alábbi matematikai képleteket C nyelvű kifejezéssé!

$$w = \frac{a \quad b + c \cdot d}{b - c}$$

$$z = \frac{\frac{x}{y} + \frac{w}{x} \cdot m - 1}{\frac{a}{b} + 2 \quad d}$$

3.5. A C nyelv utasításai

A C nyelven megírt program végrehajtható része elvégzendő tevékenységekből (utasításokból) épül fel. A C nyelv utasításainak többsége alapvetően megegyezik más magasszintű programozási nyelvek, mint például a Pascal vagy az Ada utasításaival. Az utasítások a strukturált programozás alapelveinek megfelelően ciklusok, programelágazások és vezérlésátadások szervezését teszik lehetővé. A C nyelv más nyelvekhez hasonlóan rendelkezik a vezérlésátadás **goto** utasításával, melynek használata nehezen követhetővé teszi a program szövegét. Ezen utasítás használata azonban az esetek többségében elkerülhető a **break** és a **continue** utasítások bevezetésével.

A C szabvány hét csoportban osztályozza a C nyelv utasításait:

Kifejezés utasítás	
Üres utasítás:	;
Összetett utasítás:	{}
Szelekciós utasítások:	**if** **else** **switch**
Címkézett utasítások:	**case** **default** *ugrási címke*
Vezérlésátadó utasítások:	**break** **continue** **goto** **return**
Iterációs (ciklus) utasítások:	**do** **for** **while**

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

3.5.1. Utasítások és blokkok

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

```
kifejezés;
```

A *kifejezés utasítás* végrehajtása a kifejezésnek, a 3.4. fejezetben ismertetett szabályok szerint történő kiértékelését jelenti. Mielőtt a következő utasításra kerülne a vezérlés, a teljes kiértékelés (a mellékhatásokkal együtt) végbemegy. Nézzünk néhány kifejezés utasítást:

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

Az *üres utasítás* egyetlen pontosvesszőből áll:

```
;
```

Az üres utasítás használatára akkor van szükség, amikor logikailag nem kívánunk semmilyen tevékenységet végrehajtani, azonban a szintaktikai szabályok szerint a program adott pontján utasításnak kell szerepelnie. Az üres utasítást, melynek végrehajtásakor semmi sem történik, gyakran használjuk a **do**, **for**, **while** és **if** szerkezetekben.

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

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

a következő három esetben használunk:

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

Felhívjuk a figyelmet arra, hogy blokkot nem kell pontosvesszővel lezárni. A következő példa összetett utasításában felcseréljük az *a* és *b* változók értékét, amennyiben *a* nagyobb mint *b*:

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

3.5.2. Az if utasítás

A C nyelv két lehetőséget biztosít a program kódjának feltételhez kötött végrehajtására. Ez a két mechanizmus az **if** és a **switch** utasítások használatával építhető be a programunkba. Az **if** utasítással, ami a C nyelv egyik legegyszerűbb vezérlési szerkezete, ebben az alfejezetben foglalkozunk, míg a **switch** utasítást a következő rész tárgyalja.

Az **if** utasítás segítségével valamely tevékenység (*utasítás*) végrehajtását egy *kifejezés* (feltétel) értékétől tehetjük függővé. Az **if** alábbi formájában az *utasítás* csak akkor hajtódik végre, ha a *kifejezés* értéke nem nulla (igaz):

```
if (kifejezés) utasítás
```

A C programokban a fenti utasítást jobban olvasható alakban szokás használni. A továbbiakban mindkét alakot feltüntetjük az utasítás formájának bemutatásakor:

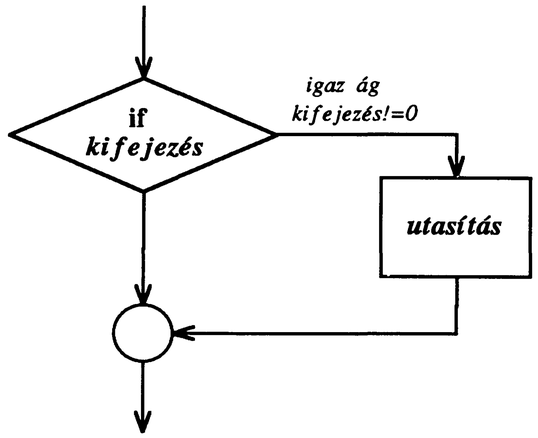
```
if (kifejezés)
   utasítás
```

A következő példaprogram egyetlen karaktert olvas a billentyűzetről, ha a karakter az *escape* (<Esc>) karakter, akkor kilépés előtt hangjelzést ad. Ha

nem az <Esc > billentyűt nyomjuk le, a program egyszerűen befejezi a futását (IF1.C).

```
#include <stdio.h>
#include <conio.h>

#define ESC 27
main()
{
  char ch;
  printf("Kérek egy karaktert: ");
  ch=getch();
  if (ch == ESC)
     printf("\aEsc\n");
}
```

A különböző vezérlési szerkezetek működésének grafikus szemléltetésére a blokkdiagramot szokás használni. Az egyszerű **if** utasítás feldolgozását a 3.13. ábrán követhetjük nyomon. (Az ábrázolás egyszerűsítése céljából feltételeztük, hogy a blokkdiagram rombusz elemében a kifejezés kiértékelése és 0-val történő összevetése egyaránt végbemegy.)



3 13 ábra
*Az **if** utasítás*

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

```
if (kifejezés != 0)
```


helyett az

```
if (kifejezés)
```

használjuk. Ez a megoldás általában világos, de bizonyos esetekben rejtélyesnek tűnhet. Külön felhívjuk a figyelmet arra, hogy a feltétel kifejezést körülvevő zárójelet mindig ki kell tenni.

3.5.2.1. Az if-else szerkezet

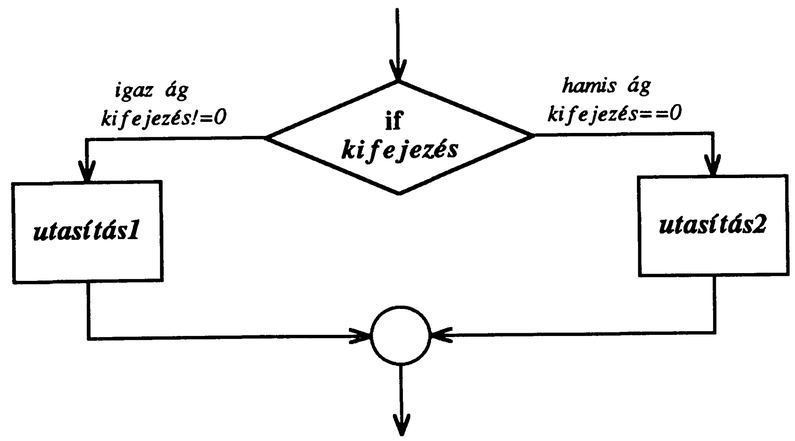
Az **if** utasítás teljes formájában, amely tartalmazza az **else**-ágat, arra az esetre is megadhatunk egy tevékenységet (*utasítás2*), amikor a *kifejezés* (feltétel) értéke zérus (hamis):

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

vagy az áttekinthetőbb alak:

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

Az *if-else* konstrukció logikai vázlata a 3.14. ábrán látható. Ha az *utasítás1* és az *utasítás2* nem összetett utasítások, akkor pontosvesszővel kell őket lezárni.



3 14 ábra
*Az **if-else** utasítás*

Az alábbi példában a beolvasott egész számról **if** utasítás segítségével döntjük el, hogy az páros vagy páratlan (IF2.C):

```
#include <stdio.h>

main()
{
  int n;

  printf("Kérek egy egész számot: ");
  scanf("%d", &n);
  if (n % 2 == 0)
     printf("A szám páros!\n");
  else
     printf("A szám páratlan!\n");
}
```

Az **if** utasítások egymásba is ágyazhatók. Ilyenkor azonban óvatosan kell eljárnunk, hisz a fordító nem mindig úgy értelmezi az utasítást, ahogy mi gondoljuk. Az alábbi példában azt várjuk az **if**-es szerkezettől, hogy a megadott egész számról megmondja, hogy az negatív páros szám-e, vagy nem negatív szám. A megoldás

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

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

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

A másik járható út, ha a belső **if** utasítást kapcsos zárójelek közé, azaz utasítás blokkba helyezzük:


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

3.5.2.2. Az else-if szerkezet

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

```
if (kifejezés)
   utasítás
else if (kifejezés)
   utasítás
else if (kifejezés)
   utasítás
else
   utasítás
```

Ezzel a szerkezettel a program többirányú elágaztatását valósíthatjuk meg. Ha bármelyik *kifejezés* igaz, akkor a hozzákapcsolt *utasítás* kerül végrehajtásra. Amennyiben egyik *feltétel* sem teljesült, a program végrehajtása az utolsó **else** utasítással folytatódik. Bizonyos esetekben nincs szükség alapértelmezés szerinti tevékenység végrehajtására - ekkor az

```
else
   utasítás
```

elhagyható. Külön kiemelést érdemel az **elsi-if** programrészlet olvashatósága.

Az alábbi példában az *n* számról eldöntjük, hogy az negatív, nulla vagy pozitív. A megoldás első alakjában elhagytuk az utolsó **else**-ágat,

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

míg a második esetben alapértelmezés szerint a számokat negatívnak tekintjük.

```
if (n > 0)
     printf("Pozitív szám\n");
else if (n==0)
     printf("Nulla\n");
else
     printf("Negatív szám\n");
```

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

```
12.45+34.55
```

alakban kell megadni (szóköz nem használható a számok és a műveleti jel között). A programban nem vizsgáljuk nullával való osztást, ennek kezelését a futtató rendszerre bíztuk.

```
#include <stdio.h>
#include <stdlib.h>
#define PROMPT ':'

main()
{
  double a,b,e;
  char op;
  putchar(PROMPT);                  /* a készenléti jel */
  scanf("%lf%c%lf", &a, &op, &b);/* a beolvasás      */

  if (op == '+')                    /* összeadás ?      */
       e = a + b;
  else if (op == '-')               /* kivonás ?        */
       e = a - b;
  else if (op == '*')               /* szorzás ?        */
       e = a * b;
  else if (op == '/')               /* osztás ?         */
       e = a / b;
  else                              /* hibás művelet!   */
       {
         printf("Hibás művelet!\n");
         exit(-1);                  /* kilépés a programból */
       }
  printf("%.2lf %c %.2lf = %.3lf\n",a,op,b,e);
}
```


Az ilyen esetekben, amikor egész értékek egyenlősége alapján ágaztatjuk el a programot, a **switch** utasítás sokkal áttekinthetőbb megoldási lehetőséget biztosít.

3.5.3. A switch utasítás

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

```
switch (kifejezés)
{
   case konstans kifejezés :
              utasítások

   case konstans kifejezés :
              utasítások

   default :
              utasítások
}
```

A **switch** utasítás először kiértékeli a *kifejezést*, majd átadja a vezérlést arra a **case** címkére (esetre), amelyben a *konstans kifejezés* értéke megegyezik a kiértékelt *kifejezés* értékével - a program futása ettől a ponttól folytatódik. Amennyiben egyik **case** konstans sem egyezik meg a *kifejezés* értékével, a program futása a **default** címkével megjelölt utasítástól folytatódik. Ha nem használunk **default** címkét, akkor a vezérlés a **switch** utasítás blokkját záró } utáni utasításra adódik.

Amikor a **case** vagy a **default** címkével belépünk a **switch** utasítás törzsébe, akkor attól a ponttól kezdve a megadott utasítások sorban végrehajtódnak (a blokkot záró zárójel eléréséig). Általában azonban az adott esethez tartozó programrészlet végrehajtása után a **goto**, a **break** vagy a **return** utasítással kilépünk a **switch** utasításból. Amennyiben a **switch** utasítás után álló utasítással kívánjuk folytatni a program futását, akkor a **break** utasítást használjuk.

A **switch** utasításban megadható esetek száma implementációfüggő - az ANSI C szabvány legalább 257 esetet javasol. Minden egyes esethez tartozó konstans kifejezésnek egyedi (nem ismétlődő) értékkel kell rendelkeznie.

A **switch** használatára tipikus példa az előző alfejezet kalkulátor programjának átírt változata (SWCALC.C):

```
#include <stdio.h>

#define PROMPT ':'

main()
{
  double a,b,e;
  char op;

  putchar(PROMPT);                  /* a készenléti jel */
  scanf("%lf%c%lf", &a, &op, &b);/* a beolvasás      */

  switch (op)
  {
    case '+' :                      /* összeadás ?      */
       e = a + b;
       break;
    case '-' :                      /* kivonás ?        */
       e = a - b;
       break;
    case '*' :                      /* szorzás ?        */
       e = a * b;
       break;
    case '/' :                      /* osztás ?         */
       e = a / b;
       break;
    default:
       {
         printf("Hibás művelet!\n");
         return -1;                 /* kilépés a programból */
       }
  } /* A switch utasítás vége */

  printf("%.2lf %c %.2lf = %.3lf\n",a,op,b,e);
}
```

A **return** utasítással, mind a **switch** utasításból, mind pedig a ***main***() függvényből kilépünk. Általában a **default** címke utáni utasításokat is **break**-

kel zárjuk, azon egyszerű oknál fogva, hogy a **default** bárhol elhelyezkedhet a **switch** utasításon belül.

Az alábbiakban példák sorával szemléltetjük a **switch** utasítás alkalmazásának gazdag tárházát.

Az első (SWITCH1.C) példában azt mutatjuk be, hogyan lehet több esethez ugyanazt a programrészletet rendelni. A programban a válaszként beolvasott karaktert feldolgozó **switch** utasításban az *'i'* és *'I'*, illetve az *'n'* és *'N'* esetekhez tartozó **case** címkéket egymás után helyeztük el.

```
#include <stdio.h>

main()
{
  char valasz;

  printf("A válasz [I/N]?");
  valasz=getchar();                 /* karakter input */

  switch (valasz)
  {
    case 'i':  case 'I':            /* Igen ?   */
       printf("A válasz IGEN.\n");
       break;

    case 'n':                       /* Nem ?    */
    case 'N':
       printf("A válasz NEM.\n");
       break;

    default:
       {
       printf("Hibás válasz!\n");
       }
  }
}
```

A következő példában (SWITCH2.C) *0* és *7* közé eső egész számok faktoriálisát határozzuk meg a **switch** utasítás segítségével (*n!=n*(n-1) . 2*1*). A megfelelő **case** címkénél belépve, az egymás utáni műveletek során áll össze a kívánt szorzat, amit az utolsó esetnél ki is íratunk. A példában a **default** címkét a **case** címkék előtt helyeztük el, ezért mindenképpen használnunk kell a **break** utasítást a hibaüzenet kiírása után.

```
#include <stdio.h>

main()
{
  int num, fakt=1;

  printf("Kérek egy egész számot (0 - 7) : ");
  scanf("%d", &num);

  switch (num)
  {
    default:
       printf("Hibás szám: %d\n", num);
       break;
    case 7:   fakt *=7;
    case 6:   fakt *=6;
    case 5:   fakt *=5;
    case 4:   fakt *=4;
    case 3:   fakt *=3;
    case 2:   fakt *=2;
    case 0:
    case 1:
       printf("%d! = %d\n", num, fakt);
       break;
  }
}
```

3.5.4. A ciklus utasítások

A programozási nyelveken bizonyos utasítások automatikus ismétlését biztosító programszerkezetet iterációnak vagy ciklusnak (*loop*) nevezzük. Ez az ismétlés mindaddig tart, amíg az ismétlési feltétel igaznak bizonyul. A C nyelv három fajta ciklus utasítást tartalmaz, melyek formája:

while (*kifejezés*) *utasítás*

for (*kifejezés1*$_{opt}$; *kifejezés2*$_{opt}$; *kifejezés3*$_{opt}$) *utasítás*

do *utasítás* **while** (*kifejezés*)

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

A ciklusokat csoportosíthatjuk a vezérlő feltétel kiértékelésének helye alapján. Azokat a ciklusokat, amelyeknél az *utasítás* végrehajtása előtt kerül

feldolgozásra a vezérlő feltétel, *elöltesztelő* ciklusnak nevezzük. Ezeknél a ciklus soronkövetkező iterációja csak akkor hajtódik végre, ha a feltétel igaz (nem nulla). A **while** és a **for** elöltesztelő ciklusok.

Ezzel szemben a **do** ciklus legalább egyszer mindig lefut, hisz a vezérlő feltétel ellenőrzése az *utasítás* végrehajtása után történik - *hátultesztelő* ciklus.

Mindhárom ciklusfajta esetén a helyesen szervezett ciklus befejezi a működését, amikor a vezérlő feltétel hamissá (nulla) válik. Vannak esetek azonban, amikor szándékosan vagy véletlenül olyan ciklust hozunk létre, melynek vezérlő feltétele soha sem lesz hamis. Ezeket a ciklusokat *végtelen ciklusnak* nevezzük. Nézzünk néhány példát végtelen ciklusra:

```
for (;;) utasítás;
while (1) utasítás;
do utasítás while (1);
```

A ciklusokból a vezérlő feltétel hamis értékének bekövetkezte előtt is ki lehet ugrani (a végtelen ciklusból is). Erre a célra további utasításokat biztosít a C nyelv, mint a **break**, a **return**, és a ciklus törzsén kívülre irányuló **goto**.

A ciklusból nemcsak kiugrani lehet. A **continue** utasítás felhasználásával a ciklus következő iterációjával folytatódik a program futása.

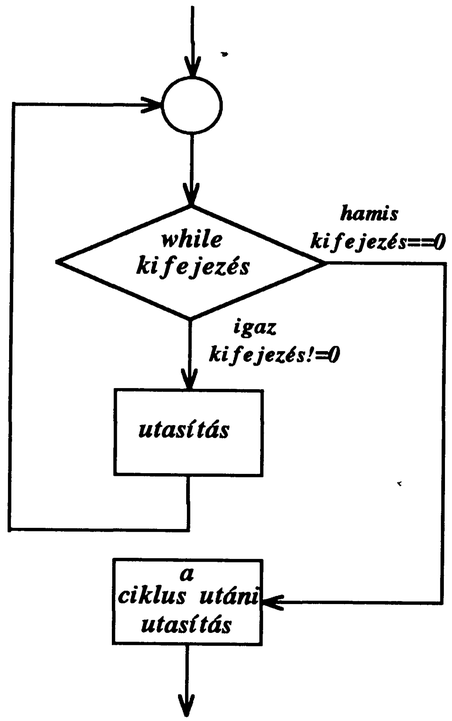
Most pedig térjünk rá az egyes ciklusfajták részletes tárgyalására.

3.5.4.1. A while ciklus

A **while** ciklus mindaddig ismétli a hozzá tartozó *utasítást* (a ciklus törzsét), amíg a vizsgált *kifejezés* (vezérlő feltétel) értéke igaz (nem nulla). A vizsgálat mindig megelőzi az *utasítás* végrehajtását. A **while** jól olvasható alakja:

```
while (kifejezés)
       utasítás
```

A **while** ciklus működésének folyamata a 3.15. ábrán követhető nyomon.



3.15 ábra
*A **while** ciklus*

A példaprogramban meghatározzuk az első *n* egész szám összegét (WHILE1:C):

```
#include <stdio.h>

main()
{
  long sum;
  int  n = 1994;

  printf("Az elsõ %d egész ", n);
  sum=0;
  while (n>0) {
     sum += n;
     n--;
  }
  printf("összege: %ld\n", sum);
}
```


A következő példában a **while** utasításba ágyazott **switch** utasítás segítségével megszámoljuk a begépelt szöveg karaktereit, szavait és sorait. A program futása a *<Ctrl+Z>* és *<Enter>* billentyűk segítségével állítható le (WHILE2.C):

```
#include <stdio.h>
#define IGEN 1
#define NEM  0

main()
{
  int ch, sor, szo, szoban=NEM;
  long karakter;

  karakter = szo = sor = 0;
  /* karakterek olvasása a Ctlr+Z és Enter lenyomásáig */
  while ((ch = getchar()) != EOF)
  {
     karakter++;
     switch (ch)
     {
       case '\n' :       /* új sor karakter */
            sor++;
       case ' '  :       /* szóelválasztó karakterek */
       case '\t' :
       case ','  :
          szoban = NEM;
          break;
       default:          /* szót alkotó karakterek */
          if (!szoban) {
             szoban = IGEN;
             szo++;
          }
          break;
     }
  }
  printf("Statisztika: %d sor, %d szó %ld karakter\n",
          sor, szo, karakter);
}
```

A **while** ciklus feltételében álló kifejezéshez hasonlóakat gyakran használunk a C nyelvű programokban:

```
(ch = getchar()) != EOF
```

Az összetett kifejezés kiértékelése a ***getchar*** könyvtári függvény meghívásával kezdődik, mely által visszaadott értéket a *ch* változó veszi fel. Majd ez az érték, a zárójelezett kifejezés értékeként, nem egyenlő relációba kerül az ***EOF*** előredefiniált konstans (*-1*) értékével. A *ch* változót **int** típussal definiáltuk, hiszen az összes lehetséges karakterkód mellett, a -1 érték is tárolására is alkalmasnak kell lennie. Hasonló módon írhatunk fel egy olyan ciklust, amely az <*Esc*> billentyű lenyomásáig várakozik:

```
while ( (ch = getch()) != 27);
```

3.5.4.2. A for ciklus

A **for** utasítást általában akkor használjuk, ha a ciklusmagban megadott utasítást adott számszor kívánjuk végrehajtani. A **for** utasítás általános alakjában külön megjelöltük az egyes kifejezések szerepét:

```
for (init_kif ; feltétel_kif ; léptető_kif)
     utasítás
```

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

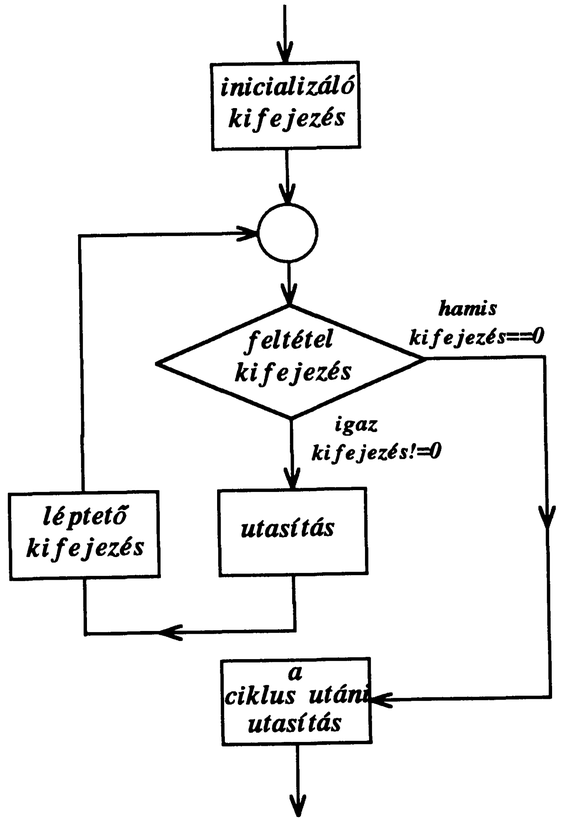
```
init_kif;
while (feltétel_kif) {
     utasítás
     léptető_kif;
}
```

Megjegyezzük azonban, hogy a két ciklus eltérően viselkedik, ha a ciklus törzsében a később ismertetésre kerülő **continue** utasítást használjuk. A for ciklus működési vázlatát a 3.16. ábra tartalmazza. Az ábrán jól nyomonkövethetők a for ciklus végrehajtásának lépései:

1 Megtörténik az *init_kif* kifejezés (amennyiben megadtuk) kiértékelése, melynek során a ciklusban használt objektumokat inicializáljuk. Semmilyen megkötés nincs az *init_kif* típusára vonatkozóan.

2. A következő lépésben a *feltétel_kif* (aritmetikai vagy mutató típusú) kifejezést dolgozza fel a rendszer (ha megadtuk). A *feltétel_kif* függvényében a ciklus az alábbi három eset valamelyikének megfelelően működhet:

- Ha a *feltétel_kif* értéke igaz (nem nulla), akkor végrehajtódik az *utasítás* melyet a *léptető_kif* kifejezés kiértékelése követ. Minden további iterációt a *feltétel_kif* kiértékelése nyit meg, és a *léptető_kif* kiértékelése zár.
- Ha a *feltétel_kif* nincs megadva, akkor annak értékét igaznak tételezi fel a rendszer és a ciklus futása pontosan megegyezik az előző esetnél tárgyalttal. Ebben az esetben a ciklus leállítása csak a **break**, a **return** és a **goto** utasítások segítségével oldható meg.
- Ha a *feltétel_kif* értéke hamis (0), akkor a **for** utasítás befejezi működését és a vezérlés a program következő utasítására kerül.



3 16 ábra
*A **for** ciklus*

Példaként a **for** ciklusra, írjuk át a **while** ciklussal megoldott, egész számok összegét meghatározó programot. Azonnal látható, hogy a megoldásnak ez a változata sokkal áttekinthetőbb és egyszerűbb (FOR1.C).

```
#include <stdio.h>

main()
{
  long sum;
  int  i,n = 1994;

  for (i=1, sum=0 ; i<=n ; i++)
      sum += i;
  printf("Az első %d egész összege: %ld\n", n, sum);
}
```

A példában szereplő ciklus magjában csak egy kifejezés utasítás található, ezért az alábbi tömörítési lépések elvégezhetők.

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

Mivel a kívánt műveletet a **for** utasítás fejében végeztük el, a ciklus törzse csak egy üres utasítást (;) tartalmaz. Azonban a fenti utasítást tovább tömöríthetjük nem kis fejtörést okozva ezzel a Pascal nyelvet ismerő barátainknak (de saját magunknak is.).

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

A ciklusokat egymásba is ágyazhatjuk, hisz a ciklus utasítása újabb ciklus utasítás is lehet. A következő példa, amelyben kettős ciklust használunk, az általános iskola első osztályában használható a szorzótábla kikérdezésére. A program az egyes táblák után leállítható az '*n*' vagy az '*N*' karakterek begépelésével, de a 9 tábla végigkérdezése után automatikusan befejezi működését. Ezt a működést egyrészről a külső ciklus feltétele

```
i < 10 && kesz != 'N'
```

másrészről pedig a *"Tovább?"* kérdésre adott válaszkarakter beolvasása és nagybetűssé való átalakítása

```
kesz = toupper(getche());
```

teszi lehetővé. Most pedig nézzük magát a (FOR2.C) programot:

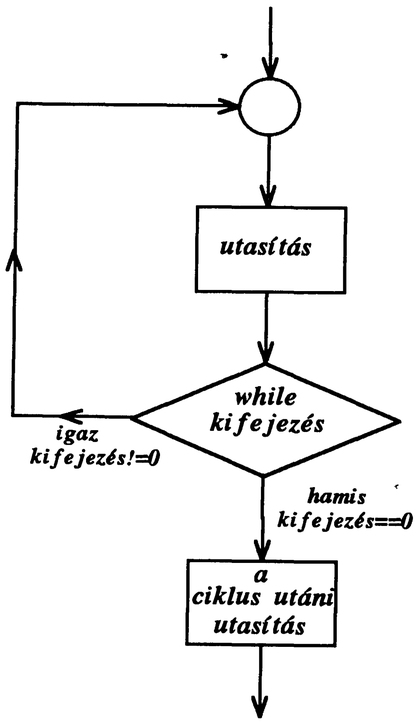

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>

main()
{
  int i, j, valasz;
  char kesz = ' ';

  printf("\nSzorzótábla program\n\n");
  for (i = 1; i < 10 && kesz != 'N'; i++)
  {
    for (j = 1; j < 10; j++)
    {
       printf ("Mennyi %d * %d ? ",i ,j);
       scanf("%d", &valasz);
       if (valasz != i * j)
           printf("hibás\n");
       else
           printf("helyes\n");
    }
    printf("\nTovább? ");
    kesz = toupper(getche());
    printf("\n");
  }
}
```

3.5.4.3. A do-while ciklus

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

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

A **do-while** ciklus futása során mindig az *utasítás* végrehajtását követi a *kifejezés* kiértékelése. Amennyiben a *kifejezés* értéke igaz (nem 0), akkor új iteráció kezdődik, míg hamis (0) érték esetén a ciklus befejezi működését. A **do-while** ciklus működését a 3.17. ábrán blokkdiagram segítségével ábrázoltuk.



3 17 ábra
*A **do-while** ciklus*

A gyakorlat azt mutatja, hogy a **do-while** ciklust sokkal ritkábban használjuk, mint a **for** és a **while** ciklusokat.
Első példaként az egész számokat összegző programnak írjuk meg a **do-while** ciklust használó változatát. Gyakorlatilag semmit sem kell megváltoztatnunk, a ciklus átszervezésén kívül (DO1.C):

```
#include <stdio.h>
main()
{
  long sum = 0;
  int  n   = 1994;

  printf("Az első %d egész ", n);
  do {
     sum += n;
     n--;
  } while (n>0);
  printf("összege: %ld\n", sum);
}
```


A második példánk egy játékprogram, amelyet futtatva a felhasználónak 0 és 100 közötti számot kell kitalálnia. A program **do-while** ciklusa egy **else-if** szerkezetet is tartalmaz, amely a felhasználó válaszát értékeli (DO2.C).

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

main()
{
  int szam, tipp, lepes = 0;

  randomize();
  szam = rand() % 101;
  printf("Gondoltam egy számot 0 és 100 között...\n");

  do {
     printf("Tipp: ");
     scanf("%d", &tipp);
     if (tipp == szam)
        printf("\nKitalálta, a gondolt szám %d !\n", szam);
     else if (tipp > szam)
        printf("..Nem talált, túl nagy!\n");
     else
        printf("..Nem talált, túl kicsi!\n");
     lepes++;
  } while (tipp != szam);

  printf("Gratulálok, %d lépésben sikerült!\n", lepes);
}
```

A példában a 0 és 100 közé eső gondolt szám előállítását a pszeudovéletlen számot szolgáltató ***rand*** függvénnyel végezzük:

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

Ahhoz azonban, hogy minden futáskor új véletlen szám keletkezzen, a véletlen szám generátort véletlenszerűen inicializáljuk a ***randomize*** hívás segítségével.

3.5.5. A break és a continue utasítások

Vannak esetek amikor egy ciklus szokásos működésébe közvetlenül be kell avatkoznunk. Ilyen feladat például, amikor adott feltétel teljesülése esetén ki

kell ugrani a ciklusból, vagy amikor a ciklus végrehajtását a következő iterációval kívánjuk folytatni.

Ezen feladatok elvégzésére a legtöbb programozási nyelv a **goto** utasítás használatára hagyatkozik, azonban a C nyelven külön utasítások - a **break** és a **continue** - állnak a programozó rendelkezésére.

3.5.5.1. A break utasítás

A **break** utasítás hatására az utasítást tartalmazó legközelebbi **switch**, **while**, **for** és **do-while** utasítások működése megszakad és a vezérlés a megszakított utasítás utáni első utasításra kerül. Az alábbi ciklus akkor lép ki, ha megtalálja a legnagyobb olyan 1994-nél kisebb egész számot, amely 123-mal osztható:

```
#include <stdio.h>
main()
{
  int i=1994, n;

   while (i>0) {
      if (!(i % 123))
          break;
      i--;
    }
    if (i) {
        n = i;
        printf("%d\n", n);
    }
}
```

A **break switch** utasításban történő felhasználását már bemutattuk 3.5.3. fejezetben, most csak a ciklusból való kiugrásra mutatunk példákat.

Az első példában (BREAK1.C) egy konstans sztringre mutató pointert addig léptetünk végig a sztringen, míg szóköztől eltérő karaktert nem találunk. Ezzel a megoldással a szöveget bevezető szóközöket kiiktathatjuk a sztringből, így a program végén kiíratva a *p* által mutatott sztringet a szóközök nem jelennek meg. A ciklus akkor is leáll, ha a sztringen végighaladva nem találunk szóköztől eltérő karaktert. Ezt a `*p!='\0'` feltétel biztosítja.


```
#include <stdio.h>

main()
{
  char *q = "      A C programozási nyelv";
  char *p;

  for (p = q; *p!='\0'; p++)
       if (*p != ' ') break;

  printf("%s\n", p);
}
```

Felhívjuk a figyelmet arra, hogy egymásba ágyazott ciklus és **switch** utasítások esetén, mindig csak a legbelső utasításból lép ki a **break** utasítás. Az alábbi kis program (BREAK2.C) kettős **for** ciklus segítségével derékszögű háromszög alakban írja ki a decimális számjegyeket:

```
#include <stdio.h>

main()
{
  int i,j;

  for (i = 1; i<10; i++)
  {
     printf("\n");
     for (j=0;;j++)
     {
        if (i == j) break;
        printf("%3d", i);
     }
  }
  printf("\n");
}
```

3.5.5.2. A continue utasítás

A **continue** utasítás a **while**, a **for** és a **do-while** ciklusok soron következő iterációját indítja el, a ciklustörzsben a **continue** után elhelyezkedő utasítások átlépésével. A **while** és a **do-while** utasítások esetén a következő iteráció a vezérlő feltétel újrakiértékelésével kezdődik. A **for** ciklus esetében azonban, a feltétel kifejezés kiértékelését megelőzi a léptető kifejezés feldolgozása.

A következő példában a **continue** segítségével értük el, hogy a *0*-tól *100*-ig egyesével lépkedő ciklusban csak a 7-tel vagy 11-gyel osztható számok kerüljenek kiírásra.

```
#include <stdio.h>
main()
{
  int i;

  for (i=0; i<100; i++) {
     if (i % 7 && i % 11)
        continue;
     printf("%d %d\n", i, i*i);
  }
}
```

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

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

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

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

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

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

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


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

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

3.5.6. A goto utasítás

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

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

```
azonosító: utasítás
```

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

```
goto azonosító;
```

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

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

```
for ( ... ) {
  for ( ... ) {
       if ( ... ) goto stop;
       . . .
  }
}
stop:
     printf("Hiba van a programban!\");
```

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

```
kesz = 0;
for ( ... ) {
  for ( ... ) {
       if ( ... ) {
            kesz = 1;
            break;
       }
       . . .
  }
  if (kesz) break;
}
printf("Hiba van a programban!\");
```

3.5.7. A return utasítás

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

A **return** utasítást

```
return ;
```

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

```
return kifejezés ;
```

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

Ellenőrző kérdések

1. Csoportosítsa a C nyelv utasításait!
2. Hogyan lehet több utasítást egyetlen utasítássá összekapcsolni?

3. Mire szolgálnak a feltételes utasítások?
4. Milyen formában használható az **if** utasítás?
5. Milyen lehetőséget biztosít a C nyelv a többirányú elágaztatás megvalósítására?
6. Mi a szerepe a **switch** utasításban a **default** címkének?
7. Milyen programszerkezetek szolgálnak az utasítások ismételt végrehajtására?
8. Melyek az előltesztelő és a hátultesztelő ciklus utasítások?
9. Hogyan lehet kiugrani a ciklus utasításokból?
10. Mikor használjuk a **break** utasítást a **switch** utasításban?
11. Mire szolgál a **continue** utasítás?
12. Mit ír ki a következő programrészlet?

```
int a;
for (a = 0; a < 10; a++);
     printf("%d\n", a);
```

Feladatok

1. Írjon programot a víz halmazállapotának vizsgálatára, amely a víz hőmérsékletét olvassa be és *víz*, *jég* vagy *gőz* szöveget írja ki a hőmérséklettől függően. (FELT1.C)

2. 0 és 100 pont közötti osztályzatok kiértékelése. Olvassa be a program a pontszámot és a pontszámtól függően az alábbi osztályzatot írja ki (FELT2.C):

Pontszám	*Osztályzat*
< 49	elégtelen
50 . 59	elégséges
60 .. 69	közepes
70 .. 88	jó
89 .. 100	jeles

3. A program olvassa be az érdemjegyet (1, 2, 3, 4, 5) számmal és írja vissza szövegesen a vizsga eredményét, használja a **switch** utasítást! (JEGY.C)

4. Írjon programot, amely 1-2000-ig kiírja a római számokat! (ROMAI.C)

5. Tanulmányozza a gyökvonás ciklussal való megoldását! (GYOK.C)

3.6. Tömbök, sztringek és mutatók

Az eddigi példákban olyan változókat használtunk, amelyek csak egyetlen érték (skalár) tárolására alkalmasak. A programozás során azonban gyakran van arra szükség, hogy azonos típusú elemekből álló adathalmazt a memóriában tároljunk és az adatokon valamilyen műveletet hajtsunk végre. (Ezt a feladatot elvileg egyedi változók sorával is meg lehet oldani, azonban ilyen tárolási mód mellett a feldolgozás programozása a programozók legszörnyűbb rémálmaiban jön csak elő.)

A C nyelven, hasonlóan más programozási nyelvekhez, tömb létrehozásával elegánsan megoldhatjuk a fenti problémát. A tömb (*array*) típus olyan objektumok halmaza, amelyek azonos típusúak és a memóriában folytonosan helyezkednek el. A tömb elemeinek típusa a **void** és a függvénytípus kivételével tetszőleges típus lehet. Az elemek elérése a tömb nevét követő idexelés operátorban megadott elemsorszám (index) segítségével történik. A tömb tehát a változók olyan készlete, melyekre közös névvel és egy indexszel hivatkozunk.

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

3.6.1. Egydimenziós tömbök

Az egydimenziós tömböket deklarálnunk kell, melynek általános alakja:

```
típus tömbnév[méret];
```

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

Általánosan elmondható, hogy *T* típus esetén a *T[méret]* típus - *méret* darab *T* típusú elem tárolására alkalmas - egydimenziós tömb (vektor) típusa. Az elemek indexelése *0*-tól (*méret-1*)-ig történik.

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

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

A tömb elemeinek egymás után történő elérésére általában a **for** ciklust használjuk, melynek változója a tömb indexe:

```
int i;

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

A tömb elemeire pedig az indexelés operátorának (*[]*) segítségével hivatkozunk. A 3.18. ábrán felvázoltuk a tömb elhelyezkedését a memóriában. Minden elem esetén feltüntettük az elem értékét a ciklus lefutása után, és az elemre való hivatkozás módját.

3 18 ábra
Az a tömb tárolása a memóriában

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

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

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

```
double nap[24];

nap[-1] = 12.34;              /* !hiba! */
nap[24] = 356.23;             /* !hiba! */
```

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

A következő (ATLAG.C) példában a 4-elemű vektorba beolvasott számokat átlagoljuk, majd kiírjuk az átlagot és az egyes elemek eltérését ettől az átlagtól. Gyakori megoldás, hogy a tömb méretét makro segítségével adjuk meg. Így, ha a méretet később meg kell változtatnunk, akkor csak egy helyen kell a változtatást elvégezni.

```
#include <stdio.h>
#define NUM 4

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

 for (i = 0; i < NUM; i++)    /* Az elemek beolvasása */
 {
     printf("szam[%d] = ", i);
     scanf("%lf", &szam[i]);
     atlag += szam[i];        /* Az elemek összegzése */
 }

 atlag /= NUM;                /* Az átlag kiszámítása  */
 printf("\nAz átlag: %lf\n\n",atlag);

 for (i = 0; i < NUM; i++)    /* Az eltérések kiírása  */
     printf("%d.\t%lf\t%lf\n", i, szam[i], atlag-szam[i]);
}
```


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

```
szam[0] = 1.3
szam[1] = 2.1
szam[2] = 4.5
szam[3] = 2.6

Az átlag: 2.625000

0.      1.300000        1.325000
1.      2.100000        0.525000
2.      4.500000        -1.875000
3.      2.600000        0.025000
```

3.6.1.1. Egydimenziós tömbök inicializálása

A C nyelv lehetővé teszi, hogy a tömböket a definiálásuk során konstans értékekkel inicializáljuk. Ez a kezdőértékadás eltér az egyszerű változók esetén használt megoldástól:

```
típus tömbnév[méret] = { vesszővel tagolt konstansok };
```

Nézzünk néhány példát vektorok inicializálására:

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

char szo[8] = { 'a', 'l', 'm', 'a'};

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

Ez első példában az *a* tömb minden elemét elláttuk kezdőértékkel. Ha több elemet adunk meg, mint amennyi a tömb mérete, akkor fordítási hibaüzenetet kapunk.

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

Az utolsó példában a *szamok* tömb elemeinek számát az inicializációs listában megadott konstansok számának (4) megfelelően állítja be a fordítóprogram. Jól használható ez a megoldás, ha a tömb elemeit fordításonként változtatjuk.

A kérdés csak az, hogyan tudjuk meg a programon belül a tömb méretét? A választ az elemszámok előzőekben bemutatott meghatározása adja meg:

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

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

vagy

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

3.6.1.2. Egydimenziós tömbök és a typedef

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

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

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

vagy

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

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

A (VEKTOR.C) példaprogramban az *a* és *b* vektorokat konstansokkal inicializált vektorként definiáltuk:

```
#include <stdio.h>

typedef int vektor8[8];

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



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

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

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

 printf("\nd = ( ");         /* A d vektor kiírása       */
 for (i = 0; i < 8; i++)
     printf(i < 8-1 ? "%d, " : "%d ", d[i]);
 printf(")\n");
}
```

3.6.2. Mutatók és a tömbök

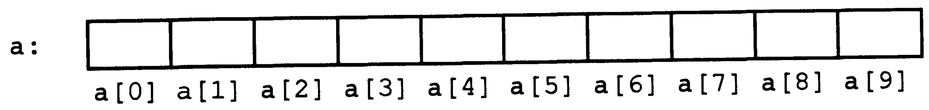
A C nyelvben a mutatók és a tömbök között szoros rokoni kapcsolat van. Ezért általában a tömböket és a mutatókat együtt szokták tárgyalni. Minden művelet, ami tömb indexeléssel elvégezhető, mutatók segítségével szintén megvalósítható. A mutatókat használó megoldás általában gyorsabb, csak éppen sokkal nehezebb megérteni.

Már most a fejezet legelején felhívjuk a figyelmet arra, hogy míg az egydimenziós tömbök (vektorok) és az egyszeres indirektségű mutatók között 100%-os (tartalmi és formai) az analógia, addig a többdimenziós tömbök és a többszörös indirektségű mutatók között ez a kapcsolat csak formai.

Nézzük meg, honnan származik ez a vektorok és az egyszeres indirektségű mutatók között fennálló szoros kapcsolat. Definiáljunk egy 10 elemű egész vektort:

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

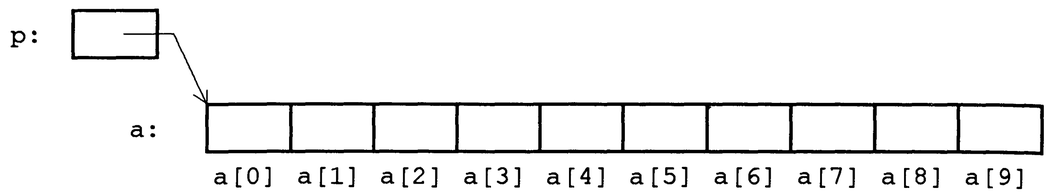
A vektor elemei a memóriában adott címtől kezdve folytonosan helyezkednek el. Mindegyik elemre `a[i]` formában hivatkozhatunk:



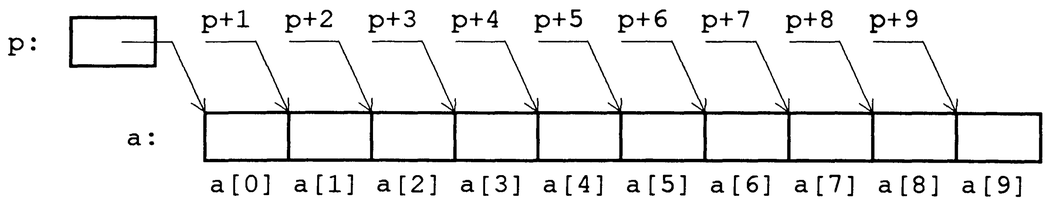
Most vegyünk fel egy *p* egészre mutató pointert, majd a "címe" operátor segítségével állítsuk az *a* tömb elejére (a 0. elemére):

```
int *p;

p = &a[0];
```

Ezek után, ha hivatkozunk a *p* mutató által kijelölt (*p) objektumra, akkor valójában az a[0] elemre hivatkozunk:



Ha *p* memória objektumra mutat, akkor a mutató aritmetika szabályai alapján a p+1, a p+2 , stb. címek az adott objektum után elhelyezkedő objektumokat jelölik ki. (Megjegyezzük, hogy negatív számokkal az objektumot megelőző elemeket címezhetjük meg.) Ennek alapján a *(p+i) kifejezéssel a tömb minden elemét elérhetjük:



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

```
p = a;          p++;
```


Az *a* tömbnevet használva azonban mindkét művelet esetén hibajelzést kapunk:

```
a = p;          a++;        /* !!! hiba !!! */
```

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

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

vagy

```
p = a;
```

akkor azonosak a következő, egy sorban elhelyezkedő, hivatkozások:

– A tömb i-dik elemének címe:

```
&a[i]     &p[i]     a+i       p+i
```

– A tömb 0-dik eleme:

```
a[0]      p[0]      *a        *p        *(a+0)    *(p+0)
```

– A tömb i-dik eleme:

```
a[i]      p[i]      *(a+i)    *(p+i)
```

A C fordító az a[i] hivatkozásokat automatikusan *(a+i) alakúra konvertálja, majd ezt a pointeres alakot lefordítja. Az analógia azonban visszafelé is igaz, vagyis az indirektség (*) operátora helyett mindig használhatjuk az indexelés ([]) operátorát. A többdimenziós tömbök és a mutatók kapcsolatával a későbbiekben foglalkozunk.

3.6.3. Sztringek

Az egydimenziós tömböket leggyakrabban karakter sztringek létrehozására használjuk. A C nyelv nem rendelkezik önálló sztring típussal, ezért a karakter tömböket használja a sztringek tárolására. A sztring tehát olyan karakter (**char**) tömb, melyben a karaktersorozat végét nulla értékű byte ('\0') jelzi.

Nézzük meg, hogyan tárolja a *"C-nyelv"* sztringet a C nyelv:

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

Amikor helyet foglalunk valamely sztring számára, akkor a sztring végét jelző byte-ot is figyelembe kell venni.

Ha az *str* tömbben maximálisan 80 karakteres sztringeket szeretnénk tárolni, akkor a tömb méretét 80+1=81-nek kell megadnunk:

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

A programozás során gyakran használunk kezdőértékkel ellátott sztringeket. A kezdőérték megadására használható a vektoroknál bemutatott megoldás, azonban a '\0' karakter megadásáról nem szabad megfeledkeznünk:

```
char st1[10] = { 'A', 'L', 'M', 'A', '\0' };

char st2[] = { 'A', 'L', 'M', 'A', '\0' };
```

Az *st1* sztring számára 10 byte helyet foglal a fordító és az első 5 byte-ba bemásolja a megadott karaktereket. Az *st2* azonban pontosan annyi byte hosszú lesz, ahány karaktert megadtunk az inicializációs listában.

A karakter tömbök inicializálása azonban sokkal biztonságosabban elvégezhető a sztring literálok (sztring konstansok) felhasználásával:

```
char st1[10] = "ALMA";

char st2[] = "ALMA";
```

A kezdőértékadás ugyan mindkét esetben - a karakterenkénti és a sztring konstanssal elvégzett inicializálás- azonos eredményt szolgáltat, azonban a sztring konstans használata sokkal áttekinthetőbb. Nem beszélve arról, hogy a sztringeket lezáró 0 byte-ot szintén a fordító helyezi el a sztringben.

A sztringek kezelésére karakter mutatókat is szoktunk használni, azonban a mutatókkal óvatosan kell bánnunk. Tekintsük az alábbi gyakran használt definíciókat. Első esetben a fordító létrehozza a 16 elemű *s* tömböt, majd oda bemásolja az inicializáló sztring karaktereit és a '\0' karaktert. A

második esetben a fordító az inicializáló sztringet eltárolja a sztring literálok számára fenntartott területen, majd a sztring kezdőcímével inicializálja a létrejövő *ps* mutatót.

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

char * ps = "gamma";
```

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

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

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

```
s = "iota";   /* !!! hiba !!! /
```

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

Művelet	*Függvény*
sztring beolvasása	***scanf, gets***
sztring kiírása	***printf, puts***
értékadás	***strcpy***
hozzáfűzés	***strcat***
sztring hosszának lekérdezése	***strlen***
sztringek összehasonlítása	***strcmp***

(Sztringkezelő függvények használata esetén a STRING.H *include* állományt be kell építenünk a forrásprogramba.)

Az alábbi (UPPER.C) példaprogram a beolvasott szöveget nagybetűssé konvertálva, fordítva írja vissza a képernyőre. A példából jól látható, hogy a sztringek hatékony kezelése érdekében a könyvtári függvényeket és a sztring karaktertömb értelmezését szintén felhasználjuk.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

main()
{
 char s[80];
 int  i;

 printf("Kérek egy szöveget: ");
 gets(s);

 for (i = strlen(s)-1; i >= 0; i--)
     printf("%c", toupper(s[i]) );
 printf("\n");
}
```

Amennyiben egy sztringen végig kell lépkedni karakterenként, akkor választhatunk a tömbös és a pointeres megközelítés között. A következő (TITKOS.C) programban a beolvasott sztringet először titkosítjuk a kizáró vagy művelet felhasználásával, majd visszaállítjuk az eredeti tartalmát. (A titkosításnál a karaktertömb, míg a visszakódolásnál a mutató értelmezést használjuk.)

```
#include <stdio.h>

#define KULCS 0xE7;

main()
{
  char s[80], *p;
  int i;

  printf("Kérek egy szöveget    : ");
  gets(s);

  for (i = 0; s[i]; i++)          /* A titkosítás    */
      s[i] ^= KULCS;
  printf("A titkosított szöveg : %s\n", s);

  p=s;
  while (*p)                      /* A visszaállítás */
     *p++ ^= KULCS;
  printf("Az eredeti szöveg    : %s\n", s);
}
```


Mind a két esetben a ciklusok leállási feltétele a sztringet záró 0-ás byte elérése volt.

3.6.4. Többdimenziós tömbök

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

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

ahol dimenziónként kell megmondani a méreteket. (A dimenziók számára semmilyen korlátozást nem tartalmaz a C nyelv definíciója.) A gyakorlatban a többdimenziós tömbök helyett általában mutató tömböket használunk.

Könyvünkben csak a kétdimenziós tömbök bemutatására és használatára szorítkozunk, azonban a kétdimenziós tömbök (mátrixok) ismeretében a megoldások több dimenzióra is általánosíthatók.

Nézzük meg először a kétdimenziós tömbök általános definícióját:

```
típus tömbnév[méret1][méret2];
```

ahol az első dimenzió (*méret1*) a tömb sorainak, míg a második (*méret2*) a tömb oszlopainak számát határozza meg. A kétdimenziós tömb megértésében sokat segíthet a matematikában használt mátrixok ismerete, amely egyben a kétdimenziós tömbök leggyakoribb felhasználási területe is.

Első lépésként tároljuk az alábbi 3x5-ös, egész elemeket tartalmazó mátrixot, a C program megfelelő objektumában:

$$\begin{bmatrix} 4 & 26 & 90 & 14 & 11 \\ 13 & 30 & 70 & 63 & 9 \\ 7 & 87 & 60 & 19 & 12 \end{bmatrix}$$

A definícióban kezdőértékként megadhatjuk a mátrix elemeit, csak arra kell ügyelnünk, hogy sorfolytonos legyen a megadás. A legegyszerűbb módszer, amikor egyszerűen soronként, vesszővel elválasztva felsoroljuk az elemeket. Az ellenőrzés gyors elvégzése érdekében ajánlott a mátrix formát betartani:

```
int matrix[3][5] = { 4, 26, 90, 14, 11,
                     13, 30, 70, 63,  9,
                      7, 87, 60, 19, 12 };
```

A másik megadási módszer során külön kihangsúlyozzuk, hogy mely elemek mely sorában tartoznak a tömbnek, hisz a sorokat külön zárójelezzük:

```
int matrix[3][5] = { { 4, 26, 90, 14, 11 },
                     {13, 30, 70, 63,  9 },
                     { 7, 87, 60, 19, 12 } };
```

Ezzel a módszerrel olyan inicializálást is megadhatunk, ahol a sorokban csak az első néhány elem kap kezdőértéket.

A sikeres tárolás után végezzünk műveletet a kétdimenziós tömb elemein, például keressük meg az elemek közül a maximálisat és a minimálisat. A kétdimenziós tömb bejárása kettős for ciklussal egyszerűen megoldható. A tömb elemeinek eléréséhez az indexelés operátorát használjuk méghozzá kétszer. A

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

hivatkozással a 2. sor 3. sorszámú elemét (19) jelöljük ki. Nem szabad megfeledkeznünk arról, hogy az indexek értéke minden dimenzióban 0-val kezdődik. A 3.19. ábrán a *matrix* kétdimenziós tömb elemei mellett feltüntettük a sorok és az oszlopok (*s/o*) indexeit is.

A sorok címe:		s/o	0	1	2	3	4
matrix[0]	→	0	4	26	90	14	11
matrix[1]	→	1	13	30	70	63	9
matrix[2]	→	2	7	87	60	19	12

3 19 ábra
A mátrix tömb tárolása

A táblázat első oszlopában található kifejezések, amelyeket a második dimenzió elhagyásával kapunk, a tömb sorainak kezdőcímét tartalmazzák. Nem kell tehát csodálkoznunk azon a megállapításon, hogy a kétdimenziós tömb a C nyelven egy olyan vektor (egydimenziós tömb), melynek elemei vektorok (mutatók). Ennek ellenére a többdimenziós tömbök mindig

folytonos memóriaterületen helyezkednek el. A példánkban a mátrix sorai képezik azokat a vektorokat, amelyekből a *matrix* vektor felépül

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

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

Térjünk vissza eredeti feladatunkhoz, a legkisebb és a legnagyobb elem kiválasztásához, melynek megoldását az alábbi programrészlet tartalmazza:

```
int emax=-1000, emin = 1000;
int i,j;

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

Végezetül jelenítsük meg a *matrix* nevű kétdimenziós tömb tartalmát a képernyőn, méghozzá mátrixos alakban. A kiíratáshoz szintén a fenti kettős ciklust használjuk. A belső (*j*) ciklus kiírja a sorokat, míg a külső (*i*) ciklus a sorok között lépked:

```
for (i = 0; i < 3; i++)
{
   for (j = 0; j < 5; j++)
       printf("%d\t", matrix[i][j]);
   printf("\n");
}
```

3.6.5. Mutatótömbök, sztringtömbök

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

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

Tekintsük az alábbi két definíciót:

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

int *b[5];
```

Formailag az a[2][5] és a b[2][5] hivatkozások egyaránt helyesek, hiszen mindkét esetben egyetlen **int** típusú elemet jelentenek

Azonban *a* egy igazi kétdimenziós tömb, amely számára a fordító 50 **int** típusú elem tárolására alkalmas helyet foglal le. Az elemek helyének meghatározására a fordító az alábbi hagyományos formulát használja:

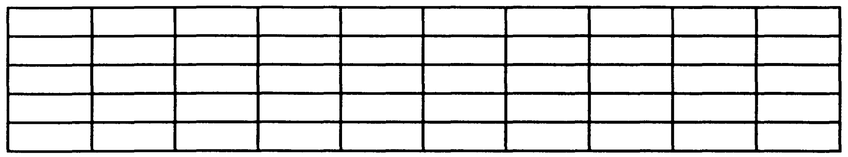
```
10 * sor + oszlop
```

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

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

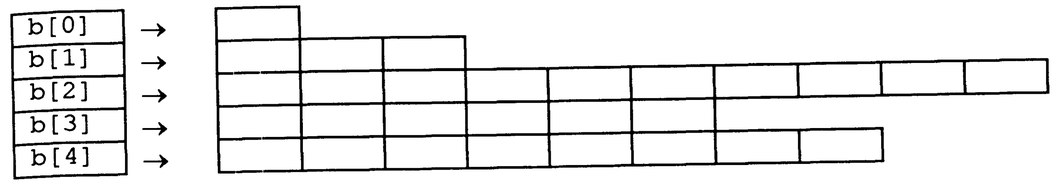
(A **static** tárolási osztályt azért kellett használnunk, mert csak a globális tömbök címe ismert a fordítás folyamán.)

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



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

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



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

A sztringtömböket általában kezdőértékek megadásával definiáljuk. Nézzünk példát sztringtömbök kialakítására kétdimenziós karaktertömb, illetve karakter típusú mutatókat tartalmazó pointertömb felhasználásával:

```
char nyar1[][10] = { "?????",
                     "Június",
                     "Július",
                     "Augusztus" };
```

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

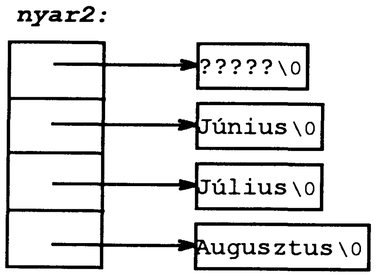
nyar1:

?????\0	Június\0	Július\0	Augusztus\0
0	10	20	30

A második esetben mutatótömböt használunk a nyári hónapok neveinek tárolására. Érdemes összehasonlítani a két megoldást mind a definíció, mind pedig a memóriahasználat szempontjából. A sztring tömb definíciója,

```
char *nyar2[]  = { "?????",
                   "Június",
                   "Július",
                   "Augusztus" };
```

amellyel a memóriában négy különálló területet foglaltunk le és ezeket a területeket teljesen feltöltötte a fordító:



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

```
printf("Hónap : %s\n",nyar1[3]);
printf("Hónap : %s\n",nyar2[3]);

printf("A hónap első betűje: %c\n",nyar1[ 3][ 0] );
printf("A hónap első betűje: %c\n",nyar2[ 3][ 0] );
```

A képernyőn az alábbi sorok jelennek meg a kiírások végrehajtása után:

```
Hónap : Augusztus
Hónap : Augusztus
A hónap első betűje: A
A hónap első betűje: A
```

3.6.6. Dinamikus helyfoglalású tömbök

A tömböket használó program fejlesztése során nagyon hamar memóriakorlátokba ütközhetünk. Ezért a C programban a nagyobb tömbök létrehozását és felszabadítását nem bízzuk a fordítóra, hanem a dinamikus memóriakezelés lehetőségeit kihasználva mi gondoskodunk ezen műveletek elvégzéséről. Mint már említettük, a dinamikus memóriahasználat alapelve az, hogy az objektum számára csak akkor foglalunk helyet, amikor szükségünk van az objektumra, illetve ha már nincs szükségünk rá, akkor az általa elfoglalt memóriaterületet felszabadítjuk.

C nyelven a dinamikus memóriakezeléshez mutatókat kell használnunk. A tömbök és mutatók közötti (tartalmi és formai) analógia tisztázása után nincs

akadálya annak, hogy tömbök számára dinamikusan foglaljunk memóriaterületet. A foglalás a már bemutatott ***malloc*** függvénnyel is elvégezhetjük, azonban tömbök esetén a ***calloc*** függvény használata javasolt. A ***calloc*** a lefoglalt területet 0-ás byte-okkal fel is tölti, ami főleg sztringek esetén hasznos. A dinamikus memóriafoglaló és -felszabadító (***free***) függvények deklarációját az STDLIB.H állomány tartalmazza.

A dinamikus helyfoglalású tömbök használata előtt vessünk egy pillantást a ***calloc*** függvény deklarációjára:

```
void * calloc( size_t elemszam, size_t elemmeret);
```

A függvény első argumentuma az elemek számát, a második pedig egy elem méretét definiálja. A függvényérték a lefoglalt területre mutató általános pointer (**void** *), illetve nulla (***NULL***), ha a foglalást nem sikerült végrehajtani.

3.6.6.1. Dinamikusan létrehozott vektorok

Először ismerkedjünk meg a leggyakrabban használt tömbfajta - az egydimenziós tömbök - dinamikus kezelésével. Induljunk ki egy szokványos tömbdefinícióból. A feladat az, hogy dinamikus memóriakezeléssel helyettesítsük az alábbi tömbdefiníciót:

```
double dt[1000];
```

Ehhez a szabad memórián kívül, csupán egy mutatóra és a ***calloc*** függvény használatára van szükség:

```
double *pd;

pd = (void *) calloc( 1000, sizeof(double) );
```

Természetesen nem szabad megfeledkezni a *pd* mutató értékének vizsgálatáról, hogy eldöntsük, sikeres volt-e a helyfoglalás. (A vizsgálat elvégzése a példában tanulmányozható.) Felhívjuk a figyelmet a tömbdefiníció és a ***calloc*** függvény paraméterezése közötti kapcsolatra.

A lefoglalt terület valójában egy mutatóval kijelölt blokk a memóriában. A blokkban tárolt **double** típusú objektumok elérése azonban a mutatók és a

vektorok közötti analógia alapján nem jelenthet gondot. Sőt választhatunk is a pointeres és a tömbös felfogás között:

```
*(pd + i)          pd[i]
```

Ha már nincs a tömbre szükségünk, nem szabad megfeledkeznünk a lefoglalt terület felszabadításáról:

```
free( pd );
```

Az alábbi (DVEKTOR.C) példában a lefoglalt területet feltöltjük véletlen értékekkel, majd Shell-rendezéssel sorbarendezzük a tömb elemeit:

```
#include <stdio.h>
#include <stdlib.h>
#define NELEM 1000

main()
{
  double *pd, sv;
  int i , j, lepes;

  /* Helyfoglalás ellenőrzéssel */
  pd = (double *) calloc( NELEM, sizeof(double));
  if (! pd) {
     printf("\a\nNincs elég memória!\n");
     return -1;      /* Hiba történt */
  }

  /* A tömb feltöltése véletlen számokkal */
  for (i = 0; i < NELEM; i++)
      *(pd + i) = random(10000)*12.34;

  /* Shell - rendezés */
  for (lepes = NELEM/2; lepes >0;  lepes /=2)
     for (i = lepes; i < NELEM; i++)
       for (j=i-lepes; j>=0 && pd[ j]>pd[ j+lepes] ; j-=lepes)
        {
            sv          =  pd[ j] ;
            pd[ j]      =  pd[ j+lepes] ;
            pd[ j+lepes]=  sv;
        }

  /* A lefoglalt terület felszabadítása */
  free(pd);

  return 0;          /* Hibátlan volt a program futása */
}
```


3.6.6.2. Kétdimenziós tömbök dinamikus kezelése

Kétdimenziós tömbök kialakítására az előző alfejezetekben két megoldást láttunk. Az első esetben igazi tömbről beszélünk, abban az értelemben, hogy a tömb elemei egyetlen folytonos területen helyezkednek el. Az ilyen tömbök definiálása után

```
double dm[10][100];
```

a fordító a `dm[i][j]` elem helyét a memóriában a *i*100+j* formula alapján határozza meg. (Felhívjuk a figyelmet arra, hogy ebben az összefüggésben a sorok száma nem játszik szerepet.)

A másik esetben ún. mutatótömböt használtunk, ahol a mutatótömb és az egyes sorok önmagukban folytonos területen helyezkednek el, de ez nem mondható el a teljes adatstruktúráról.:

```
double *pd[10];
```

Mindkét megoldás esetén több lehetőség közül is választhatunk, abból a szempontból, hogy a kétdimenziós tömbünket teljesen vagy csak részlegesen dinamikus memóriafoglalással hozzuk létre.

A kétdimenziós tömbre mutató pointer használata

Tekintsük az alábbi két definíciót:

```
int *p1[10];        int (*p2)[10];
```

Az első alak ismerős, hiszen a *p1* 10-elemű **int** * mutatókat tartalmazó tömb. A második definíció értelmezésénél tekintsük egyetlen egységnek a zárójelben található kifejezést, amely ily módon 10 elemű egész vektort határoz meg. Mivel azonban a zárójelben mutató áll, a *p2* nem más, mint 10 elemű egész tömbre mutató pointer.

A definíciók azonnal értelmezhetők lesznek, ha a **typedef** tárolási osztályt használjuk a felírásukhoz:

```
typedef int * iptr;          typedef int ivektor10[10];
iptr p1[10];                 ivektor10 *p2;
```

Hasonló megfontolással definiáljuk az 50x100-as **double** típusú elemeket tartalmazó kétdimenziós tömb típusát:

```
typedef double  matrix [ 50 ] [ 100 ];
```

Ezek után az előző alfejezetben bemutatott megoldással (dinamikus helyfoglalással) létre tudjuk hozni a *matrix* típusú kétdimenziós tömböt. Ehhez felhasznált mutató a *pm*

```
matrix *pm;
```

amellyel a helyfoglalást végző utasítás:

```
pm = (matrix *) calloc ( 1 , sizeof(matrix) );
```

Mivel a *pm* mutató magára a mátrixra mutat, az elemekre való hivatkozás formája:

```
(*pm)[ i][ j]
```

Az alábbi (DMATRIX.C) példában két mátrixot hozunk létre, és az egyik tartalmának 10-szeresét a másikba másoljuk. (A kiindulási mátrix feltöltéséhez szintén véletlen számokat használunk.)

```
#include <stdio.h>
#include <stdlib.h>
#define NELEM1 50
#define NELEM2 100

typedef double matrix [ NELEM1] [ NELEM2] ;

main()
{
  matrix *pm1, *pm2;
  int i , j;

  /* Helyfoglalás ellenőrzéssel */
  pm1 = (matrix *) calloc( 1 , sizeof(matrix));
  pm2 = (matrix *) calloc( 1 , sizeof(matrix));
  if (!pm1 || !pm2) {
     printf("\a\nNincs elég memória!\n");
     return -1;
  }
```



```
  /* Az 1. mátrix feltöltése véletlen számokkal */
  for (i = 0; i < NELEM1; i++)
    for (j = 0; j < NELEM2; j++)
        (*pm1)[ i][ j] = random(10000)*1.234;

  /* Az 1. mátrix 10-szeresének másolása a 2. mátrixba */
  for (i = 0; i < NELEM1; i++)
    for (j = 0; j < NELEM2; j++)
        (*pm2)[ i][ j] = (*pm1)[ i][ j] * 10.0;

  /* A lefoglalt területek felszabadítása */
  free(pm1);
  free(pm2);
}
```

Kétdimenziós tömb emulációja vektorban

Talán egyik legérthetőbb megoldás többdimenziós tömbök dinamikus megvalósítására, az amikor egyetlen vektort foglalunk le, melynek mérete megegyezik a többdimenziós tömb elemszámával. A kétdimenziós tömböknél maradva az *[i][j]* indexű elem relatív távolsága mindig meghatározható a

```
d = i * oszlopok száma + j
```

összefüggés alapján. (Többdimenzió esetén ezen összefüggés kibővített formája szintén felírható.)

A fenti összefüggés felhasználásával oldjuk meg az előző mátrixos példát megfelelő méretű dinamikus vektorok segítségével (DMVEKT.C):

```
#include <stdio.h>
#include <stdlib.h>
#define NELEM1 50
#define NELEM2 100

main()
{
  double *pd1, *pd2;
  int i , j;

  /* Helyfoglalás ellenőrzéssel */
  pd1 = (double *) calloc(NELEM1 * NELEM2, sizeof(double));
  pd2 = (double *) calloc(NELEM1 * NELEM2, sizeof(double));
```

```
    if (!pd1 || !pd2) {
       printf("\a\nNincs elég memória!\n");
       return -1;
    }

    /* Az 1. mátrix feltöltése véletlen számokkal */
    for (i = 0; i < NELEM1; i++)
      for (j = 0; j < NELEM2; j++)
           pd1[ i*NELEM2 + j] = random(10000)*1.234;

    /* Az 1. mátrix 10-szeresének másolása a 2. mátrixba */
    for (i = 0; i < NELEM1; i++)
      for (j = 0; j < NELEM2; j++)
          pd2[ i*NELEM2 + j] = pd1[ i*NELEM2 + j] * 10.0;

    /* A lefoglalt területek felszabadítása */
    free(pd1);
    free(pd2);
}
```

Dinamikus helyfoglalású mutatótömbök

Ha mutató vektort használunk a kétdimenziós tömb megvalósításához, akkor további két lehetőség közül választhatunk. A

```
double *pv[50];
```

definíció hatására a fordító létrehozza a mutatókat tartalmazó *pv* 50 elemű vektort, melynek az elemei azonban sehova sem mutatnak. Ekkor adott a lehetőség, hogy a tömb sorait képező **double** típusú vektorokat egyenként dinamikusan hozzuk létre. (A vektorok hosszát az *elemszám* kifejezés helyén kell megadni.) A sorok foglalását a megfelelő ellenőrzéssel együtt ciklusban érdemes elvégezni,

```
for (i = 0; i < 50; i++) {
    pv[i] = (double *) calloc( elemszám, sizeof(double));
    if (!pv[i]) {
        printf("\a\nNincs elég memória!\n");
        return -1;
    }
}
```

A lefoglalt vektorokat egyenként kell felszabadítani, ezért ehhez a művelethez is ciklust használunk:


```
    for (i = 0; i < 50; i++)
        free(pv[i]);
```

A tömb elemeire való hivatkozás ebben az esetben formailag teljesen megegyezik a szabványos kétdimenziós tömböknél használttal. Az alábbi három alak közül bármelyik használható:

```
pv[i][j]        *(pv[i]+j)          *(*(pv+i) + j)
```

A mátrixos példa megoldása pointertömb segítségével az alábbiakban látható (PTRVEKT.C). A programban a különböző hivatkozási lehetőségeket felváltva használjuk.

```
#include <stdio.h>
#include <stdlib.h>
#define NELEM1 50
#define NELEM2 100

main()
{
  double *pv1[ NELEM1] , *pv2[ NELEM1] ;
  int i , j;

  /* Helyfoglalás ellenőrzéssel */
  for (i = 0; i < NELEM1; i++) {
      pv1[ i] = (double *) calloc( NELEM2, sizeof(double));
      pv2[ i] = (double *) calloc( NELEM2, sizeof(double));
      if (!pv1[ i] || !pv2[ i] ) {
          printf("\a\nNincs elég memória!\n");
          return -1;
      }
  }
  /* Az 1. mátrix feltöltése véletlen számokkal */
  for (i = 0; i < NELEM1; i++)
    for (j = 0; j < NELEM2; j++)
        *(pv1[ i] + j) = random(10000)*1.234;

  /* Az 1. mátrix 10-szeresének másolása a 2. mátrixba */
  for (i = 0; i < NELEM1; i++)
    for (j = 0; j < NELEM2; j++)
        pv2[ i][ j] = *( *(pv1+i) + j) * 10.0;
  /* A lefoglalt területek felszabadítása */
  for (i = 0; i < NELEM1; i++) {
      free(pv1[ i] );
      free(pv2[ i] );
  }
}
```

A mutatótömb használatának másik lehetősége, amikor a mutatókat tartalmazó tömböt is dinamikus helyfoglalással hozzuk létre. Ehhez mindössze egy

```
double **pp;
```

mutatót kell definiálnunk. A mutatót felhasználva helyet foglalunk 50 darab **double** * típusú mutató számára:

```
pp = (double **) calloc ( 50, sizeof( double *));
```

Sikeres memóriafoglalás esetén a megoldás további lépései megegyeznek a nem dinamikusan létrehozott mutatóvektornál használtakkal. Egyetlen eltérés a program végén jelentkezik, amikor a sorok felszabadítása után gondoskodnunk kell a dinamikus mutatótömb által elfoglalt terület szabaddá tételéről is:

```
free(pp);
```

A teljes (PPTRVEKT.C) példaprogram áttanulmányozása sokat segíthet a kezdő programozók szárnára kuszának tűnő deklarációk és hivatkozások megértésében:

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

#define NELEM1 50
#define NELEM2 100

main()
{
  double **pp1, **pp2;
  int i , j;

  /* Helyfoglalás ellenőrzéssel */

  /* A mutatóvektorok létrehozása */
  pp1 = (double **) calloc( NELEM1 , sizeof(double *));
  pp2 = (double **) calloc( NELEM1 , sizeof(double *));
  if (!pp1 || !pp2) {
     printf("\a\nNincs elég memória!\n");
     return -1;
  }
```



```
/* Helyfoglalás a kétdimenziós tömb sorai számára */
for (i = 0; i < NELEM1; i++) {
    pp1[ i]    = (double *) calloc(NELEM2, sizeof(double));
    *(pp2+i) = (double *) calloc(NELEM2, sizeof(double));
    if (!pp1[ i] || !pp2[ i] ) {
       printf("\a\nNincs elég memória!\n");
       return -1;
    }
}

/* Az 1. mátrix feltöltése véletlen számokkal */
for (i = 0; i < NELEM1; i++)
  for (j = 0; j < NELEM2; j++)
      *(pp1[ i] + j) = random(10000)*1.234;

/* Az 1. mátrix 10-szeresének másolása a 2. mátrixba */
for (i = 0; i < NELEM1; i++)
  for (j = 0; j < NELEM2; j++)
      pp2[ i][ j] = *( *(pp1+i) + j) * 10.0;

/* A lefoglalt területek felszabadítása */

/* A tömb sorainak felszabadítása */
for (i = 0; i < NELEM1; i++) {
    free(pp1[ i] );
    free(*(pp2+i));
}
/* A mutatóvektorok felszabadítása */
free(pp1);
free(pp2);
}
```

Ellenőrző kérdések

1. Mit nevezünk tömbnek?
2. Hogyan lehet egydimenziós tömböt létrehozni C nyelven?
3. Hogyan adunk kezdőértéket a vektornak?
4. Mi a kapcsolat a mutatók és a vektorok között?
5. Hogyan tárolja a sztringeket a C nyelv?
6. Milyen műveleteket használhatunk a sztringekkel kapcsolatosan?
7. Hogyan tárolja a C nyelv a kétdimenziós tömb elemeit a memóriában?
8. Hogyan lehet sztringeket tömbben tárolni?

9. Milyen lépéseket tartalmaz a dinamikusan helyfoglalású vektor kezelése?
10. Értelmezze az alábbi definíciókat!

```
int a[23];
int *p = a;
char t[10][80];
char *cp[10];
```

Feladatok

1. Írjon programot amely statisztikát készít a beolvasott sztringben található magánhangzókról! (MAGANH.C)

2. Írjon programot amely egy 5x5-ös mátrix főátlójába 1, míg a többi elembe 0-át ír! (EGYSEGM.C)

3. Írjon programot amely egy 10 elemű vektort feltölt egész számokkal, majd a páratlan indexű elemeket 3-mal, míg a páros indexű elemeket 12-vel megszorozza! (SZOROZ.C)

4. Készítsen egy sakktábla feltöltő programot, amely a 8x8-as tömböt dinamikusan foglalja le, és a fekete kockákat 1-gyel, a fehéreket pedig 0-val tölti fel! (SAKKT.C)

5. Írjon programot, amely beolvas egy mondatot.
   Például: `Ma szep az ido.`
   Visszaírja: `Mava szevep avaz ividovo.`
   Tehát a magánhangzó esetében v betűvel megismétli a magánhangzót:
   (i -> ivi, o -> ovo, e -> eve, a -> ava, u ->uvu).
   (VAVEVI.C)

6. Készítsen programot, amely a beolvasott (maximálisan 10 darab) nevet névsorba rendezi. (NEVSOR.C)

3.7. Felhasználó által definiált adattípusok

A C nyelv lehetővé teszi, hogy a nyelv meglévő típusait felhasználva újabb típusokat hozzunk létre. Az eddigiek folyamán már többször éltünk ezzel a lehetőséggel, amikor a **typedef** segítségével szinonim típusneveket vezettünk be. Ugyancsak ide tartoznak a felsorolt (**enum**) típusok, amelyeket csoportos, egymással kapcsolatban álló, konstansok létrehozására használunk. Az **enum** típusnév önmagában semmire sem használható, hiszen a felsorolt típust a C nyelv felhasználójának (a programozónak) kell definiálnia, az alábbi formában:

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

A felsorolt típus definíciójának formája és alkalmazásának szabályai megegyeznek a további három felhasználói típusával, amelyek azonban lényegileg különböznek az **enum** típustól.

Ebben a fejezetben mélyrehatóan foglalkozunk a struktúra, a bitmező és az unió típusokkal. A tömb és a struktúra típusokat közös néven összeállított (*aggregate*) típusoknak nevezzük. A fenti három típus közül gyakorlati szempontból a struktúrának van igazán kiemelkedő szerepe, ezért elsősorban a **struct** típusokkal foglalkozunk. A struktúra típussal kapcsolatos fogalmak és megoldások minden további nélkül alkalmazhatók a bitmező és az unió típusra is.

3.7.1. A struktúra típus megadása

Az előző fejezetben megismerkedtünk azokkal a módszerekkel, amelyek lehetővé teszik, hogy azonos típusú objektumokat logikailag egyetlen egységben a tömbben tároljunk. A programozás során azonban gyakran találkozunk olyan problémákkal, amelyek megoldásához különböző típusú objektumokat önálló programozási egységben kell feldolgoznunk. Tipikus területe az ilyen jellegű feladatoknak az adatbázis-kezelés, ahol a file tárolási egysége a rekord, tetszőleges mezőkből épülhet fel.

C nyelven a struktúra (**struct**) típus több, tetszőleges típusú (kivéve a **void** és a függvény típust) objektum együttese. Ezek az objektumok önálló, a struktúrán belül érvényes nevekkel rendelkeznek. Az objektumok szokásos

elnevezése struktúra elem vagy adattag (*member*). (A más nyelveken megszokott mező (*field*) elnevezést a később ismertetésre kerülő bitstruktúrák esetén használja a C nyelv.)

A struktúra típusú változó létrehozása logikailag két részre osztható. Először deklarálnunk kell magát a struktúra típust, melyet felhasználva változókat definiálhatunk. A struktúra szerkezetét meghatározó deklaráció általános formája:

```
struct struktúra_azonosító {
                    típus1 tag1;
                    típus2 tag2;
                       . . .
                    típusN tagN;
                    };
```

Felhívjuk a figyelmet arra, hogy a struktúra deklarációja azon kevés esetek egyike, ahol a pontosvesszőt kötelező kitenni. Az adattagok deklarációjára a C nyelv szokásos deklarációs szabályai érvényesek. A fenti típussal struktúra változót a már megismert módon készíthetünk:

```
struct struktúra_azonosító struktúra_változó;
```

A *struktúra_azonosító* a **struct** kulcsszóval együtt jelöli az új felhasználói típust.

Az általános formában bemutatott deklaráció és definíció csak egy (de talán a leggyakrabban használt) megoldás a C nyelv által biztosított lehetőségek közül.

Nézzünk egy konkrét példát a struktúra típus megadására. Könyvtárkezelő program készítése során jól alkalmazható az alábbi adatstruktúra:

```
struct book {
            char nev [ 20 ];  /* a szerző neve */
            char cim [ 40 ];  /* a mű címe     */
            int ev;           /* a kiadás éve  */
            float ar;         /* a könyv ára   */
};
```

A típusdeklaráció után változókat is létrehozhatunk:

```
struct book macska, gyerek, cprog;
```


A C-nyelv megengedi ugyan, hogy a fenti két lépést egybeépítsük, de a program áttekinthetősége érdekében ezt a megoldást érdemes kerülni:

```
struct book {
            char nev [ 20 ];  /* a szerző neve */
            char cim [ 40 ];  /* a mű címe     */
            int ev;           /* a kiadás éve  */
            float ar;         /* a könyv ára   */
} macska, gyerek, cprog;
```

A harmadik szintén elég gyakran alkalmazott megoldás a **typedef** kulcsszóra épül:

```
typedef struct book {
                      char nev [ 20 ];  /* a szerző neve */
                      char cim [ 40 ];  /* a mű címe     */
                      int ev;           /* a kiadás éve  */
                      float ar;         /* a könyv ára   */
                    } BOOK;
```

A deklaráció során keletkező mindkét típusnév (a hagyományos **struct** *book* és az új *BOOK*) ugyanazt a struktúratípust jelöli. (Ebben az esetben a **struct** kulcsszót követő *book* név el is hagyható.) A BOOK típussal szintén változókat definiálhatunk:

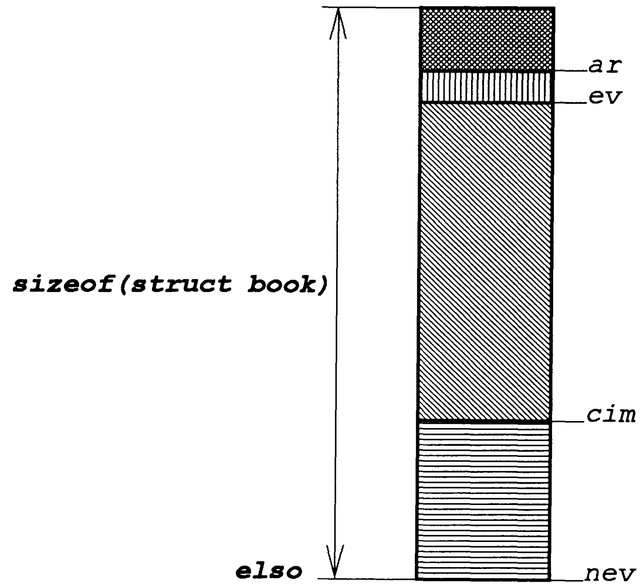
```
BOOK macska, gyerek, cprog;
```

Bármelyik lehetőséget használjuk, a lényeg az, hogy létrehoztunk egy új felhasználói típust. A struktúra típusú változó adattagjait a fordító a deklaráció sorrendjében tárolja a memóriában. A 3.20. ábrán grafikusan ábrázoltuk a

```
struct book elso;
```

definícióval létrehozott objektum felépítését. Az ábráról az is leolvasható, hogy az adattagok nevei az adattagoknak a struktúra objektum elejétől mért távolságát jelölik. A struktúra mérete általában megegyezik az adattagok méretének összegével. Vannak azonban olyan számítógépek, ahol bizonyos típusú adatok csak megadott (byte-, szó-, stb.) határon kezdődhetnek. Ezeken a számítógépeken a fenti megállapítás nem igaz, hiszen a megfelelő határra való igazítás következtében "lyukak" keletkeznek a struktúra változóban. A

sizeof operátor használatával azonban adott a lehetőség, hogy minden számítógépen a pontos méretet kapjuk meg.



3 20 ábra
*A **struct** book típus felépítése*

3.7.2. Hivatkozás a struktúra adattagjaira

A struktúra szót gyakran önállóan is használjuk, ilyenkor azonban nem a típusra, hanem az adott struktúra típussal létrehozott objektumra (változóra) gondolunk.

Az előzőekben deklarált **struct** *book* típus felhasználásával definiáljunk néhány változót:

```
struct book s1, s2, *ps ;
```

Az *s1 és s2* **struct** *book* típusú változók, amelyek tárolásához szükséges memóriaterület (**sizeof**(**struct** *book)*) lefoglalásáról a fordító gondoskodik. A *ps* pedig olyan pointer, amely **struct** *book* típusú objektumra mutathat.

Ahhoz, hogy a *ps* mutatóval is hivatkozhassunk a struktúrára, két lehetőség közül választhatunk. Az első, kevésbé hasznos esetben a *ps*-t egyszerűen ráirányítjuk az *s1* struktúrára:

```
ps = &s1;
```

A második lehetőség a dinamikus memóriafoglalás használatát jelenti. Az alábbi programrészletben helyet foglalunk a *book* struktúra számára, majd pedig felszabadítjuk azt. Természetesen a hibaellenőrzésről sem szabad megfeledkeznünk:

```
ps = (struct book *) malloc( sizeof(struct book) );
if (!ps) exit(-1);
. . .
free(ps);
```

A megfelelő definíciók elvégzése után van három struktúránk, az *s1*, *s2* és a **ps* Nézzük meg, hogyan lehet értéket adni a struktúráknak! Erre a célra a C nyelvben a pont (.) operátor használható:

```
strcpy( s1.nev, "Kernighan-Ritchie");
strcpy( s1.cim, "The C Programming Language");
s1.ev = 1988;
s1.ar = 24.95;
```

A pont operátor baloldali operandusa a struktúra objektum, a jobboldali operandus pedig a struktúrán belül jelöli ki az adattag objektumot.

A pont operátort a *ps* által mutatott objektumra alkalmazva a precedencia szabályok miatt zárójelek között kell megadni a **ps* kifejezést:

```
strcpy( (*ps).nev, "Kernighan-Ritchie");
strcpy( (*ps).cim, "The C Programming Language");
(*ps).ev = 1988;
(*ps).ar = 24.95;
```

Mivel a C nyelvben (főleg a C nyelv első változatában) gyakran használunk mutató által kijelölt struktúrákat (például függvények argumentumaként), a C nyelv ezekben az esetekben egy önálló operátort - a nyíl (->) operátort - biztosít az adattag hivatkozások elvégzésére. (A nyíl operátor két karakterből, a mínusz és a nagyobb jelből áll.) A nyíl operátor használatával olvashatóbb formában írhatjuk fel *ps* által kijelölt struktúra adattagjaira vonatkozó értékadásokat:

```
strcpy( ps->nev, "Kernighan-Ritchie");
strcpy( ps->cim, "The C Programming Language");
ps->ev = 1988;
ps->ar = 24.95;
```

A nyíl operátor baloldali operandusa a struktúra objektumra mutató pointer, míg a jobboldali operandus - a pont operátorhoz hasonlóan - a struktúrán belül jelöli ki az adattag objektumot. Ennek megfelelően a *ps->ar* kifejezés jelentése: "*a **ps** mutató által kijelölt struktúra **ar** adattagja*".

A "címe" (&) operátort az *s1* struktúrára alkalmazva szintén mutatóhoz jutunk, amellyel már használható a nyíl operátor, mint például:

```
(&s1)->ev=1988;
```

Láthatjuk, hogy a pont és a nyíl operátorok mindkét esetben - közvetlen és indirekt hivatkozás esetén - egyaránt használhatók. Szem előtt tartva a program olvashatóságát és helyességét javasoljuk, hogy a pont operátort csak közvetlen (a struktúra változó adattagjára történő) hivatkozás esetén, míg a nyíl operátort kizárólag indirekt (mutató által kijelölt struktúra adattagjára vonatkozó) hivatkozás esetén használjuk.

A struktúrára vonatkozó értékadás speciális esete, amikor egy struktúra változó tartalmát egy másik struktúra változónak kívánjuk megfeleltetni. Ezt a műveletet az eredeti C nyelven csak adattagonként lehetett elvégezni:

```
strcpy( s2.nev, s1.nev);
strcpy( s2.cim, s1.nev");
s2.ev = s1.ev;
s2.ar = s2.ar;
```

Az ***ANSI C*** szabvány azonban értelmezi a struktúra objektumra vonatkozó értékadás (=) műveletét. Ezért az alábbi értékadások az ***ANSI C*** programban érvényes kifejezéseket jelölnek:

```
 s2 = s1 ;  /* Ez felel meg a fenti 4 értékadásnak */
*ps = s2 ;
 s1 = *ps = s2 ;
```

(A szabvány azt is lehetővé teszi, hogy struktúrát közvetlenül függvény argumentumaként, illetve függvényértékként szerepeltessünk.)

Az értékadásnak ez a módja valójában egyszerűen a struktúra által lefoglalt memóriablokk átmásolását jelenti. Az értékadás művelet azonban gondot okoz akkor, amikor a struktúra olyan mutatót tartalmaz, amellyel külső memóriablokkra hivatkozunk:

```
struct string {
                char *p;
                int len;
              } st1, st2;

st1.p = "Hello";
st2   = st1;
```

A másolás végeztével a külső memóriablokk (sztring) mindkét struktúrához hozzátartozik, hiszen csak a mutatók tartalma (a blokk címe) másolódott át. A C nyelv nem tartalmaz semmilyen támogatást az ehhez hasonló problémák megoldására, ezért ezekben az esetekben az adattagonkénti értékadás módszerét ajánlott használni.

Az alábbi (STRUCT1.C) példában billentyűzetről töltjük fel a **struct** *book* típusú struktúrát, majd megjelenítjük a tárolt adatokat:

```
#include <stdio.H>

struct book {
               char nev[ 20] ;
               char cim[ 40] ;
               int ev;
               float ar;
            } ;

main()
{
   struct book  wb;

   /* Az adatok beolvasása */
   printf("\nKérem a könyv adatait!\n\n");
   printf(" Szerző      : ");    gets( wb.nev );
   printf(" Cím         : ");    gets( wb.cim );
   printf(" Kiadás éve  : ");    scanf("%d", &wb.ev );
   printf(" Ár (Ft)     : ");    scanf("%f", &wb.ar );
```

```
    /* Az adatok megjelenítése */
    printf("\nA könyv adatai :\n\n");
    printf("Szerző      :  %s\n", wb.nev );
    printf("Cím         :  %s\n", wb.cim );
    printf("Kiadás éve  :  %4d\n", wb.ev );
    printf("Ár          :  %5.2f Ft\n", wb.ar );
}
```

3.7.3. Kezdőértékadás a struktúrának

A tömbökhöz hasonlóan a struktúra definíciójában is szerepelhet kezdőértékadás. Az egyes adattagokat inicializáló konstansok vesszővel elválasztott listáját kapcsos zárójelek közé kell zárni. Példaként lássuk el kezdőértékkel **struct** *book* típusú *s1* struktúrát:

```
struct book s1 = { "Kernighan-Ritchie",
                   "The C Programming Language",
                   1988, 24.95 };
```

Amennyiben a struktúra valamely adattagja tömb, akkor a kezdőértékek listájában a tömb inicializálását végző részt külön kapcsos zárójelek között adjuk meg. A zárójelek használata nem kötelező, de az inicializálás biztonságosabbá tehető vele:

```
typedef struct {
                  int nelem;
                  int v[20];
                } VEKTOR;

VEKTOR a = {5, {1, 2, 3, 4, 5} };

VEKTOR b = {4, 10, 20, 30, 40 };
```

3.7.4. Egymásba ágyazott struktúrák

Már említettük, hogy a struktúráknak tetszőleges típusú adattagjai lehetnek. Ha egy struktúrában valamilyen más struktúra típusú adattagot használunk, ún. egymásba ágyazott struktúrát kapunk.

Tételezzük fel, hogy síkbeli geometriai objektumok adatait struktúra felhasználásával kívánjuk feldolgozni. Az alábbi struktúra a geometriai alakzat helyét meghatározó pont koordinátáinak tárolására alkalmas:


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

A kört definiáló struktúrában a kör középpontját az előzőleg deklarált *pont* adattagban tároljuk:

```
struct kor {
               struct pont kp;
               int r;
            };
```

Hozzunk létre két kört, méghozzá úgy, hogy az egyiknél használjunk kezdőértékadást, míg a másikat adattagonkénti értékadással inicializáljuk:

```
struct kor k1 = { { 100, 100 }, 50 }, k2;

k2.kp.x =  50;
k2.kp.y = 150;
k2.r    = 200;
```

A kezdőértékadásnál a belső struktúrát inicializáló konstansokat szintén nem kötelező kapcsos zárójelek közé helyezni. A *k2* struktúra adattagjait inicializáló értékadás során az első pont (.) operátorral a *k2*-ben elhelyezkedő *kp* struktúrára hivatkozunk, majd ezt követi a belső struktúra adattagjaira vonatkozó hivatkozás.

Ha a *pont* struktúrát máshol nem használjuk, akkor névtelen struktúraként közvetlenül beépíthető a *kor* struktúrába:

```
struct kor {
               struct {
                        int x;
                        int y;
                      } kp;
               int r;
            };
```

Bonyolultabb dinamikus adatszerkezetek (például lineáris lista) kialakításánál adott típusú elemeket kell láncba fűznünk. Az ilyen elemek általában valamilyen adatot és egy mutatót tartalmaznak. A C nyelv lehetővé teszi, hogy a mutatót az éppen deklarálás alatt álló struktúra típusával definiáljuk. Az ilyen struktúrákat, amelyek önmagukra mutató pointert tartalmaznak

adattagként, önhivatkozó struktúráknak nevezzük. Példaként tekintsük az alábbi *listaelem* deklarációt:

```
struct listaelem {
                   int adattag;
                   struct listaelem * kapcsolat;
                 };
```

Ez a rekurzív deklaráció mindössze annyit tesz, hogy a *kapcsolat* mutatóval az adott struktúrára mutathatunk. A fenti megoldás nem ágyazza egymásba a két struktúrát, hiszen az a struktúra, amelyre a későbbiek során a mutatóval hivatkozunk, valahol máshol fog elhelyezkedni a memóriában.

A C fordító számára a deklaráció elsősorban azért szükséges, hogy a deklarációnak megfelelően tudjon memóriát foglalni, vagyis hogy ismerje a létrehozandó objektum méretét. A fenti deklarációban a létrehozandó objektum egy mutató, amelynek mérete független a struktúra méretétől.

Mi a helyzet, ha nem egy mutatót, hanem egy *listaelem* struktúrát szeretnénk adatelemként elhelyezni a *listaelem* struktúrában?

```
struct listaelem {                                  /* !hibás! */
                   int adattag;
                   struct listaelem elem;
                 };
```

Ez a deklaráció, amennyiben azt a fordító értelmezni tudná, igazi struktúra egymásba ágyazás lenne. Mivel azonban a felhasználás helyéig, a

```
struct listaelem elem;
```

sorig, a *listaelem* struktúra mérete még nem ismert, így a fordító nem tudja feldolgozni a fenti deklarációt és a *listaelem* struktúrát definiálatlannak jelzi.

3.7.5. Struktúratömbök

Már láttunk példát arra, hogyan lehet struktúrában tömb adatelemet elhelyezni. Most azonban azt nézzük meg, hogy milyen módon lehet struktúra elemeket tartalmazó tömböket használni. Struktúratömböt pontosan ugyanúgy kell definiálni, mint bármilyen más típusú tömböt.

Példaként az előzőekben deklarált **struct** *book* típust használva hozzunk létre egy 100 kötetes "könyvtárat":

```
struct book lib[100];
```

A kérdés már csak az, hogyan tudunk hivatkozni a tömbelem struktúrák adattagjaira? Ebben az esetben a pont és az indexelés operátorát együtt kell használunk:

```
lib[13] . ar = 123.23;
```

A két operátornak azonos a precedenciája, így a kiértékelés során a balról-jobbra szabályt használja a fordító. Tehát először a tömbelem kerül kijelölésre (az indexelés), amit az adattagra való hivatkozás (pont operátor) követ. Így zárójelek használata nem szükséges, hiszen a fenti kifejezés az alábbi kifejezéssel egyenértékű:

```
( lib[13] ) . ar = 123.23;
```

A struktúratömböt a definiálásakor a szokásos módon inicializálhatjuk. A áttekinthetőség érdekében ajánlott az egyes struktúrák kezdőértékét kapcsos zárójelben elkülöníteni:

```
struct book mlib[] ={
                     { "0. Író" , "0. Könyv", 1990, 1000.0 },
                     { "1. Író" , "1. Könyv", 1991, 1001.0 },
                     { "2. Író" , "2. Könyv", 1991, 1001.0 } };
```

Amennyiben dinamikusan kívánjuk a struktúra objektumokat létrehozni, akkor mutatótömböt kell használnunk a "könyvtár" létrehozására:

```
struct book * plib[100];
```

A struktúra elemek számára az alábbi ciklus segítségével foglalhatunk helyet a dinamikusan kezelt memóriaterületen:

```
for (i = 0; i < 100; i++) {
      plib[i] = (struct book *)malloc(sizeof(struct book));
      if (!plib[i]) {
         printf("\a\nNincs elég memória!\n");
         exit(-1);
      }
}
```

A tömbelem által kijelölt struktúrára a nyíl operátor segítségével hivatkozhatunk:

```
plib[14] -> ar = 25.54;
```

Ha már nincs szükségünk a struktúra elemeire, akkor az egyes elemeken végighaladva felszabadítjuk a lefoglalt memóriaterületet:

```
for (i = 0; i < KSZAM; i++)  free(plib[i]);
```

Az alábbi (STRUCT2.C) példában a dinamikusan létrehozott, 100 könyvet tartalmazó könyvtárból, egy struktúratömbbe kigyűjtjük az 1968 és 1989 között megjelent műveket (A programnak az a része, amely a könyvtár véletlen adatokkal való feltöltését végzi, a későbbiek során file-kezelési műveletekkel helyettesíthető.)

```
#include <stdio.H>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define KSZAM 100

struct book {
            char  nev [ 20 ];
            char  cim [ 40 ];
            int   ev;
            float ar;
          };
main()
{
   struct book   talalat[ KSZAM ];
   struct book * plib[ KSZAM ];
   int i, db;

  /* Helyfoglalás ellenőrzéssel */
  for (i = 0; i < KSZAM; i++) {
      plib[ i] = (struct book *)malloc(sizeof(struct book));
      if (!plib[ i] ) {
          printf("\a\nNincs elég memória!\n");
          return -1;
      }
   }
```



```
  /* A könyvek véletlenszerű feltöltése */
  randomize();
  for (i = 0; i < KSZAM; i++) {
      sprintf(plib[ i] ->nev,"%03d Anonymous", i);
      sprintf(plib[ i] ->cim,"Nothing %03d", i);
      plib[ i] ->ev = 1900+random(100);
      plib[ i] ->ar = random(1000) * 1.5;
}

  /* A keresett kötetek kiválogatása */
  db = -1;           /* Nincs találat */
  for (i = 0; i < KSZAM; i++)
      if (plib[ i] ->ev >=1968 && plib[ i] ->ev <= 1989)
         talalat[ ++db ] = *plib[ i ];

  /* A keresés eredményének kijelzése */
if (db != -1) {
   printf("A találatok száma: %d \n\n", db+1);
   for (i=0; i<=db; i++)
   printf("%-20s %-40s %5d %10.2f\n",talalat[ i] .nev,
                                     talalat[ i] .cim,
                                     talalat[ i] .ev,
                                     talalat[ i] .ar  );
}
else
   printf("Nincs a feltételnek megfelelő könyv!\n");

   /* A lefoglalt területek felszabadítása */
   for (i = 0; i < KSZAM; i++)
       free(plib[ i] );
}
```

3.7.6. Union típusú adatstruktúrák

Nagyobb programok fejlesztése során, amikor sok adatot használunk, azon vehetjük észre magunkat, hogy betelt a memóriának az adataink tárolására kijelölt része. (Ez a probléma legélesebben a C nyelv megjelenésének idejében jelentkezett.) Ekkor felvetődik a kérdés, hogyan tudunk memóriaterületet megtakarítani? Az egyik, már többször emlegetett megoldás az, amikor a programunk bizonyos objektumait dinamikus helyfoglalással hozzuk létre. Ez napjainkban is jól használható általános orvosság a "memóriaszűke" programozási "betegség" leküzdésére.

A C nyelv kidolgozásakor további lehetőségeket is beépítettek, amelyek azonban jóval kisebb jelentőséggel bírnak, mint a dinamikus memóriakezelés.

Nézzük meg, miben áll a következő két fejezetben bemutatásra kerülő megoldások lényege:

- Helyet takarítunk meg, ha ugyanazt a memóriaterületet több objektum közösen használja (de nem egyidejűleg). Az ilyen objektumok összerendelése a C struktúra típussal rokon **union** (unió - egyesítés) típussal valósítható meg.
- A másik lehetőség, hogy az olyan objektumokat, amelyek értéke 1 byte-nál kisebb területen is elfér, egyetlen byte-ban helyezzük el. Ehhez a megoldáshoz a C nyelv a bitmezőket biztosítja. Azt, hogy milyen (hány bites) objektumok kerüljenek egymás mellé, szintén a **struct** típussal rokon bitstruktúra deklarációval lehet definiálni.

Az unió és a bitstruktúra megoldásokkal nem lehet jelentős memória megtakarítást elérni, viszont annál inkább romlik a programunk hordozhatósága. Ezért mindig érdemes megfontolni, hogy az adott esetben valóban a fenti két lehetőség valamelyike az egyetlen üdvözítő megoldás. A memóriaigény csökkentését célzó eljárások hordozható változata a dinamikus memóriafoglalás.

Napjainkban a **union** és bitstruktúra felhasználásának célja valamelyest megváltozott. Az uniót elsősorban gyors és hatékony gépfüggő adatkonverziók megvalósítására, míg a bitstruktúrát a hardver különböző elemeinek vezérlését végző parancsszavak előállítására használjuk.

Ismerkedjünk meg először a **union** típussal! Igazából nincs sok dolgunk, mivel a **struct** típussal kapcsolatban ismertetett formai megoldások, kezdve a deklarációtól, a pont és vessző operátoron át egészen az struktúratömb kialakításáig, a **union** típusra is alkalmazhatók. Egyetlen és egyben lényegi különbség az adattagok elhelyezkedése között van. Míg a struktúra adattagjai a memóriában egymás után helyezkednek el, addig az unió adattagjai közös kezdőcímen kezdődnek (átlapoltak). Készítsünk egy struktúra és egy ugyanolyan felépítésű unió deklarációt:

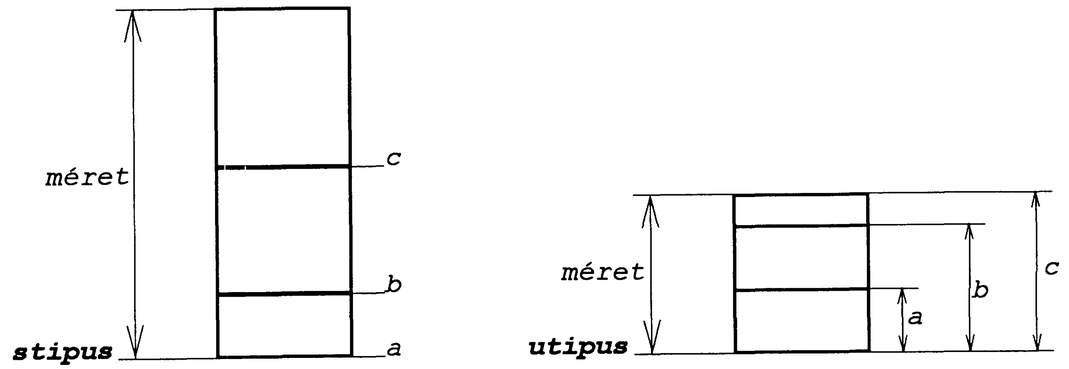
```
struct stipus
{
     int a;
     long b;
     char c[5];
};
```

```
union utipus
{
     int a;
     long b;
     char c[5];
};
```


A 3.21. ábrán felrajzoltuk mindkét esetben a memóriafoglalást. A **struct** típus méretét az adattagok összmérete (a kiigazításokkal korrigálva) adja, míg a **union** mérete megegyezik a leghosszabb adattagjának méretével.

Használva a **union** *utipus* típust, definiáljunk változókat kezdőértékkel és anélkül. Az unió inicializálásakor a kezdőértékek listája mindössze egyetlen konstanst tartalmazhat, melynek értékét mindig az unió első adattagja veszi fel.

```
union utipus u1 = { 1994 }; /* az a adattag kezdőértéke */
union utipus u2, u3;
```



3 21 ábra
*A **struct** és a **union** típusok memóriafoglalása*

Ugyancsak értelmezett a **union** objektumok közötti, és az adattagonkénti értékadás:

```
u2 = u3 = u1;
strcpy(u2.c, "alma");
u3.b = 0x41424344L;
```

A következő példában arra használjuk az unió típust, hogy az **unsigned long int** típusú adatokat leggyorsabban két **unsigned short int** típusú részre bontsuk. Először deklaráljuk a megfelelő szerkezetű **union** *convls* típust, majd hozzuk létre ezzel a típussal az *sl* változót:

```
union convls
{
     unsigned short si[2];
     unsigned long  li;
} sl;
```

További segédváltozók felvétele után az alábbi programrészlet mindkét irányú konverzióra mutat példát:

```
unsigned short hi, lo;
unsigned long  ul;

/* A long érték két részre bontása */
sl.li = 0x12345678L;  /* Ezt kívánjuk két részre bontani */
hi = sl.si[ 1] ;      /* A felső fele  : 0x1234          */
lo = sl.si[ 0] ;      /* Az alsó fele  : 0x5678          */

/* A long érték összeépítése  fordított konverzió */
sl.si[ 0] = 0x4321;   /* Az alsó fele                    */
sl.si[ 1] = 0x8765;   /* A felső fele                    */
ul = sl.li;           /* Az eredmény: 0x87654321         */
```

Az utolsó példánkban a **struct** és a **union** típus együttes használatát mutatjuk be. Sokszor szükség lehet arra, hogy egy file rekordjaiban tárolt adatok rekordonként más-más felépítésűek legyenek. Tételezzük fel, hogy minden rekord tartalmaz egy nevet és egy értéket, amely hol szöveg, hol pedig szám. A feladat megoldható egyetlen struktúratípus segítségével is, amelyben mindkét lehetséges érték számára helyet biztosítunk és a *tipus* adattag tartalmával jelöljük, hogy éppen a szám vagy a szöveg (*'d'* vagy *'c'*) az érvényes adat:

```
struct rekord_s
{
     char nev[25];
     char tipus;
     char ertek_szoveg[20];
     double ertek_szam;
};
```

Ennek a megoldásnak az a hátránya, hogy üres területeket tárolunk az adatállományban. Kevesebb lesz a kihasználatlan terület, ha a struktúrán belül unióba egyesítjük a két lehetséges értéket (variáns rekord):


```
struct rekord
{
     char nev[25];
     char tipus;
     union  {
       char szoveg[20];
       double szam;
     } ertek;
};
```

A *rekord*-ban tárolt adatok megjelenítését a következő programrészlet illusztrálja. A *tipus* adattag információjának feldolgozását a **switch** utasítás segítségével végeztük el:

```
printf("Név        :  %s\n", rek.nev);
switch (rek.tipus) {
   case 'd' :
         printf("Szám       :  %lf\n", rek.ertek.szam);
         break;
   case 'c' :
         printf("Szöveg     :  %s\n", rek.ertek.szoveg);
         break;
   default  :
         printf("Hibás adattípus!\n");
};
```

3.7.7. A bitmezők használata

A legtöbb programozási nyelvtől eltérően a C nyelv beépített módszert tartalmaz a byte-on belüli bitek elérésére. Ez a megoldás több szempontból is hasznos lehet:

- Helytakarékos, bitméretű (logikai *KI/BE*) változók használata - több változó tárolása egyetlen byte-ban,
- A hardver elemek programozásához használt bitsorozatok magasszintű kezelése.

A már megismert bitenkénti operátorok segítségével szintén elvégezhetők a szükséges műveletek, azonban a bitmezők használatával strukturáltabb és hatékonyabb kódot kapunk.

A bitmezők kezelése a bitstruktúrán keresztül valósul meg, melynek általános felépítését az alábbiakban láthatjuk:

```
struct struktúra_azonosító {
            típus  név1 : bithossz;
            típus  név2 : bithossz;
                  . . .
            típus  névN : bithossz;
            };
```

A deklarációban a bitmező neve előtt csak **unsigned int**, **signed int** vagy **int** típus szerepelhet. Ennek megfelelően a *bithossz* maximális értékét az adott számítógépen az **int** típus hossza határozza meg. A struktúra típusban a bitmezők és az adattagok vegyesen is használhatók.

Első példaként a **struct** *book* típust felhasználva egészítsük ki a könyv információkat bitméretű változók sorával:

- kölcsönözhető (1 - igen),
- kikölcsönözték (1 - igen),
- maximum hány hétre vihető el (max. 8 hét),
- a kölcsönzés dátuma (**long**).

A fenti követelményeknek megfelelő adatstruktúra deklarációja:

```
struct libbook {
   struct book konyv;
   long datum;
   unsigned kolcsonozheto : 1;
   unsigned kikolcsonozve : 1;
   unsigned het           : 4;
};
```

Kezdőértékek beállítása mellett definiáljunk egy változót ezzel a típussal:

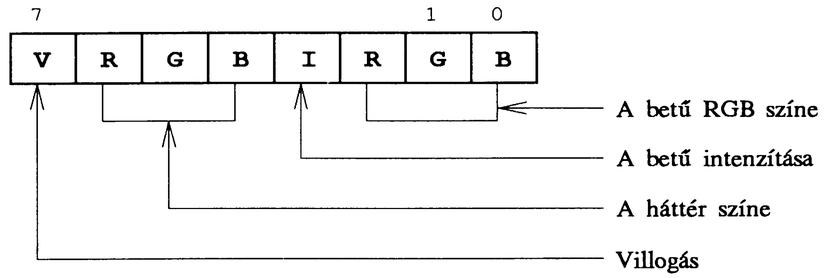
```
struct libbook cprog = { { "Kernighan-Ritchie",
                           "The C Programming Language",
                           1988, 24.95 },
                           19940813L, 1, 0, 8};
```

majd pedig adjunk értéket a bitmezőknek:


```
cprog.kikolcsonozve = 1;
cprog.kolcsonozheto = 1;
cprog.het           = 3;
```

A példából kitűnik, hogy a bitmezők segítségével egyszerűen lehet a bonyolult bitműveleteket elvégezni. A megoldás helytakarékossága szintén látható, hiszen 1 byte-ot használtunk 3 **char** típus tárolásához szükséges 3 byte helyett. Azonban az adatterület megtakarítása mellett a kódterület nagyobb lett, mintha a **char** típust használtuk volna. A futási idő szintén megnövekedett, hiszen a bitműveletek sokkal lassúbbak, mint egy **char** típusú adattagra való hivatkozás elvégzése.

A bitmezőknek az ehhez hasonló célokra való felhasználását mindig mérlegelni kell. A futásidő és a rendelkezésre álló memória- és lemezterület lehetnek a mérlegelés szempontjai.
Végezetül nézzünk egy olyan alkalmazási területet (a hardver programozása), amelynél nem vitathatók a bitmezők használatának előnyei. Tekintsük az IBM PC számítógépek színes karakteres képernyőn való megjelenítéshez használt karakter és attribútum byte-ot. Az attribútum byte egy bitstruktúra, melynek felépítése a 3.22. ábrán látható



3 22 ábra
Az attribútum byte felépítése

Az alábbi struktúra lefedi mind a karakterkódot tartalmazó byte-ot, mind pedig az attribútum byte-ot:

```
struct kepbetu {
       char kod;
       unsigned betu : 3;
       unsigned      : 1; /* Nem használjuk ezt a mezőt */
       unsigned alap : 3;
       int           : 1; /* A villogást sem használjuk */
};
```

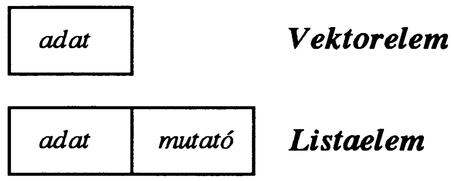
A megfelelő mutató előállítása után tegyünk ki a képernyő bal felső sarkába egy kék alapon levő piros 'A' betűt. (Felhívjuk a figyelmet arra, hogy ez a programrész nem hordozható, hisz csak IBM PC számítógépen MS-DOS alatt használható.)

```
/* Távoli (far) mutató a képernyőterület elejére */
struct kepbetu far *psc=(struct kepbetu far *)0xb0000000L;

psc -> kod  = 'A';
psc -> betu = 4;     /* piros */
psc -> alap = 1;     /* kék   */
```

Ha a bitstruktúra deklarációjában nem adunk nevet a bitmezőnek, akkor a megadott bithosszúságú területet nem tudjuk elérni (hézagpótló bitek) Amennyiben a névtelen bitmezők hosszát 0-nak adjuk meg, akkor az ezt követő adattagot (vagy bitmezőt) **int** határra igazítja a fordítóprogram.

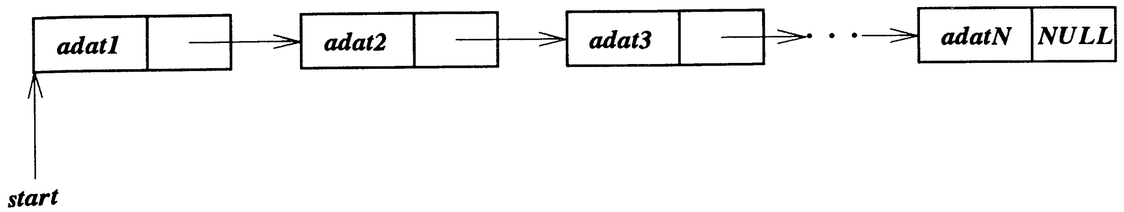
3.7.8. Önhivatkozó struktúrák használata - a listaszerkezet

A programozási munka során kitüntetett szereppel rendelkeznek azok a megoldások, amelyekkel az adatok tárolását valamilyen listaszerkezettel valósítjuk meg, a lista elemeit pedig dinamikus memóriafoglalással hozzuk létre. Nézzük meg, milyen előnyökkel jár a lista használata a vektorral (egydimenziós tömbbel) összehasonlítva. A vektor mérete definiáláskor eldől, a lista mérete azonban dinamikusan növelhető, illetve csökkenthető. Ugyancsak lényeges eltérés van az elemek beszúrása és törlése között. Míg a listában ezek a műveletek csupán néhány mutató másolását jelentik, addig a vektorban nagy mennyiségű adat mozgatását igénylik. További lényeges különbség van a tárolási egység, az elem felépítésében:



A vektor elemei csak a tárolandó adatokat tartalmazzák. Lista esetében azonban az adaton kívül a kapcsolati információk tárolására, mutató(k)ra is szükség van.

Ezen kis bevezető után ismerkedjünk meg a legegyszerűbb listaszerkezettel, a lineáris listával, melynek elemei egyirányú, egyszeres láncolással (mutatókkal) vannak összekapcsolva:



A lista azonosítására a *start* mutató szolgál, ezért ennek értékét mindig meg kell őriznünk. A lista végét nulla értékű mutatóval (***NULL***) jelezzük. A C nyelven a listaelemeket a már bemutatott önhivatkozó struktúrákkal hozhatjuk létre. Példaként tegyük önhivatkozóvá a fejezet elején deklarált **struct** *book* típust a **struct** *book* * típusú *kovetkezo* adattag bevezetésével:

```
struct book {
        char  nev [20];
        char  cim [40];
        int   ev;
        float ar;
        struct book * kovetkezo;
};
```

A következőkben áttekintjük a listakezelés alapvető műveleteit. Az itt közölt programrészek a lemezmellékleten összefüggő programként is megtalálhatók (STRUCT3.C).

A lista kezelése során szükségünk van segédváltozókra, illetve a lista kezdetét jelölő *start* mutatóra:

```
struct book  *start=NULL, *elozo, *aktualis, *kovetkezo;
```

Amikor a lista adott elemével (*aktualis*) dolgozunk, szükségünk lehet az megelőző (*elozo*) és a rákövetkező (*kovetkezo*) elemek helyének ismeretére is.

A példában 10 elemet tartalmazó listát építünk fel. A lista felépítése során minden egyes elem esetén három jól elkülöníthető tevékenységet kell végrehajtanunk:

1. helyfoglalás (ellenőrzéssel) a listaelem számára,
2. a listaelemben tárolt adatok feltöltése,
3. a listaelem hozzáfűzése a listához (a végéhez). A hozzáfűzés során az első és nem első elemek esetén más-más lépéseket kell végrehajtanunk.

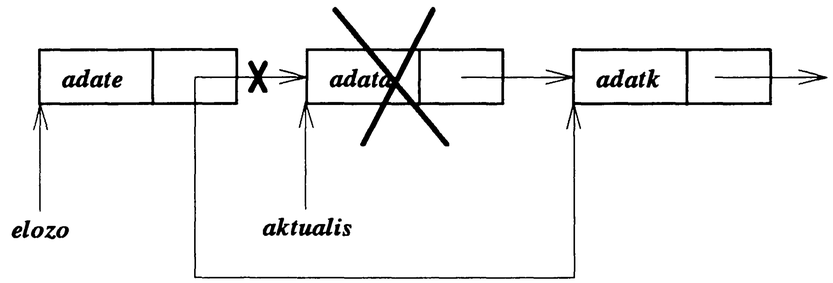
```
/* A lista felépítése és az elemek véletlen feltöltése */
randomize();
for (index = 0;index < 10;index++) {

   /* ❶ Memóriafoglalás ellenőrzéssel */
   aktualis = (struct book *)malloc(sizeof(struct book));
   if (!aktualis) {
      printf("\a\nNincs elég memória!\n");
      return -1;
   }

   /* ❷ Az listaelem adattagjainak feltöltése */
   sprintf(aktualis->nev,"%03d Anonymous", index);
   sprintf(aktualis->cim,"Nothing %03d", index);
   aktualis->ev = 1900+random(100);
   aktualis->ar = random(1000) * 1.5;
   aktualis->kovetkezo = NULL;

   /* ❸ Láncolási és léptetési műveletek */
   if (index == 0)                          /* első elem      */
      start = elozo = aktualis;
   else {                                   /* további elemek */
      /* Az új elem láncolása az előző elemhez */
      elozo->kovetkezo = aktualis;
      /* Továbblépés a listában */
      elozo = aktualis;
   }
}
```

A ciklus lefutása után kész van a 10 elemet tartalmazó listánk. Most következhetnek az ún. listakezelési műveletek. Az első ilyen művelet legyen a lista végigjárása, melynek során az egyes elemek tartalmát kiíratjuk a képernyőre. A bejárás során a *start* mutatótól indulunk és a ciklusban mindaddig lépkedünk a következő elemre, amíg el nem érjük a lista végét jelző nulla (***NULL***) pointert:


```
/* A listában lépkedve az elemek adatainak kiírása */
aktualis = start;
printf("\n");
do {
   printf("%-20s %-40s %5d %10.2f\n", aktualis->nev,
                                      aktualis->cim,
                                      aktualis->ev,
                                      aktualis->ar  );
   /* Lépés a következő elemre */
   aktualis = aktualis->kovetkezo;
}  while (aktualis != NULL);
```

Gyakran használt művelet a listaelem törlése. A példában a törlendő elemet a listaelem sorszáma alapján azonosítjuk (A sorszámozás 0-val kezdődik a ***start*** által kijelölt elemtől kezdődően - a programrészlet nem alkalmas a 0. és az utolsó elem törlésére!) A törlés művelete, melyet a 3.23. ábrán ábrázoltunk, szintén három tevékenységre tagolható:

1. az adott sorszámú elem lokalizálása a listában,
2. a törlés elvégzése,
3. a törölt elem területének felszabadítása.



3 23 ábra
Törlés a listából

A példában a 4. sorszámú elemet töröljük a listából:

```
/* A 4. sorszámú elem lokalizálása és törlése */
/* ❶ Az elem helyének meghatározása */
aktualis = start;
for (index = 0; index<4; index++) {
    elozo    = aktualis;
    aktualis = aktualis->kovetkezo;
}
```

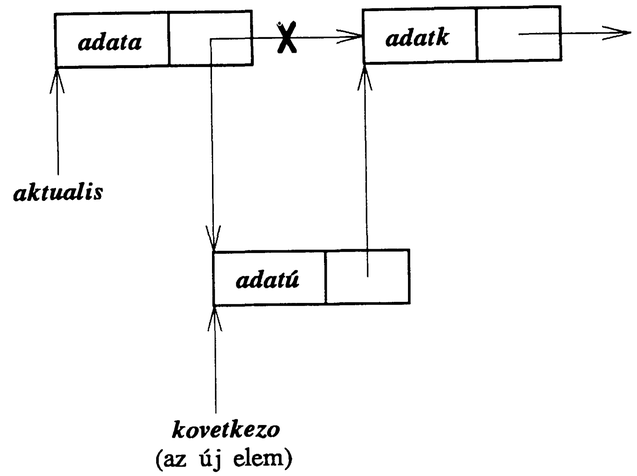
```
/* ❷ A törlés - kifűzés a láncból */
elozo->kovetkezo = aktualis->kovetkezo;
/* ❸ A terület felszabadítása */
free(aktualis);
```

A törléssel ellentétes művelet - új elem beillesztése a listába, két meglévő elem közé. A beszúrás helyét annak az elemnek a sorszámával azonosítjuk, amely mögé az új elemet beillesztjük. A beszúrás műveletét (3.24. ábra) a következő négy lépésben végezzük:

1. a beszúrás helyének lokalizálása a megelőző elem sorszáma alapján,
2. helyfoglalás az új listaelem számára,
3. a listaelem feltöltése adatokkal,
4. az elem beillesztése a sorszámmal kijelölt elem után.

A példában a 3. sorszámú elem mögé illesztünk új listaelemet:

```
/* A 3. sorszámú elem lokalizálása és mögé */
/* új elem beszúrása                       */
/* ❶ A megelőző elem helyének meghatározása */
aktualis = start;
for (index = 0; index<3; index++)
    aktualis = aktualis->kovetkezo;
/* ❷ Területfoglalás az új elem számára */
kovetkezo = (struct book *)malloc(sizeof(struct book));
if (!kovetkezo) {
    printf("\a\nNincs elég memória!\n");
    return -1;
}
/* ❸ Az új elem adatainak feltöltése */
strcpy(kovetkezo->nev,"!!! Anonymous");
strcpy(kovetkezo->cim,"Nothing !!!");
kovetkezo->ev = 1999;
kovetkezo->ar = 2222;
/* ❹ Az új elem befűzés a láncba */
kovetkezo->kovetkezo = aktualis->kovetkezo;
aktualis ->kovetkezo = kovetkezo;
```

3 24 ábra
Új elem beillesztése a listába

A listakezelő programok a fenti lépéseket használják, esetleg kibővítve az itt bemutatott lehetőségeket. Helyes programozási gyakorlat az, hogy kilépés előtt felszabadítjuk a dinamikusan foglalt memóriaterületeket. Nézzük meg, hogyan történik a lista elemeinek megszüntetése. Ebből a célból szintén végig kell mennünk a listán, ügyelve arra, hogy még az aktuális listaelem megszüntetése előtt kiolvassuk a következő elem helyét:

```
aktualis = start;
do {
    kovetkezo = aktualis->kovetkezo;
    free(aktualis);
    aktualis = kovetkezo;
} while (kovetkezo != NULL);
start = NULL; /* Nincs lista! */
```

Ellenőrző kérdések

1. Sorolja fel a C nyelv felhasználó által definiált adattípusait!
2. Milyen adattagokat tartalmazhat a struktúra típus?
3. Hogyan hivatkozunk a struktúra adattagjaira?
4. Hogyan lehet struktúra változót kezdőértékkel ellátni?
5. Mire használható a struktúratömb adattípus?

6. Milyen célra szolgál a **union** típus a C nyelvben?
7. Hogyan lehet struktúrában bitmezőket elhelyezni?
8. Mi teszi lehetővé a struktúrának listaelemként történő felhasználását?
9. Értelmezze az alábbi műveleteket!

```
struct pld { int a; double d; };
struct pld x, y = {23, 456.7};
x = y;
x.a = y.a+(int)x.d;
```

Feladatok

1. Írjon programot amely 12 elemű struktúratömb típus segítségével az alábbi szerkezetű adatok tárolására használható! (TANKOR.C)

adatok	***típus***
Név	**char** [30]
Ösztöndíj	**float**
Tankör	**int**
születési dátum:	**struct** *datum*
év	**int**
hó	**char**
nap	**char**

2. Egészítse ki a TANKOR.C programot úgy, hogy az képes legyen a legnagyobb ösztöndíjjal rendelkező diák adatait megjeleníteni a képernyőn! (TANKOR2.C)

3.8. Függvények

A függvény a C program olyan névvel ellátott egysége (alprogram), amely a program más részeiből annyiszor meghívható, ahányszor csak szükség van a függvényben definiált tevékenységsorozatra. A C program általában sok kisméretű, jól kézben tartható függvényből épül fel. A gyakran használt függvények lefordított kódját könyvtárakba rendezhetjük, amelyekből a szerkesztőprogram a hivatkozott függvényt beépíti a programunkba.

Az ANSI C szabvány függvényekkel kapcsolatosan több lényeges módosítást vezetett be. Az eredeti C nyelv ezen módosításai a megbízhatóbb programkészítést célozzák. A könyvünkben mindkét függvénymegadási formát (az eredetit és az újat) bemutatjuk, azonban a példáinkban a szabvány által ajánlott új formát használjuk.

A függvények hatékony felhasználása érdekében a C nyelv lehetőséget biztosít arra, hogy a függvény bizonyos belső objektumainak a függvényhívás során adjunk értéket. Hogy melyek ezek az objektumok és milyen típussal rendelkeznek, azt a függvény definíciójában a függvény neve után zárójelben kell megadnunk. A hívásnál pedig hasonló formában kell felsorolnunk az átadni kívánt értékeket. A szakirodalom ezekre az objektumokra és értékekre különböző nevekkel hivatkozik:

a függvény-definícióban szereplő objektumok	***a függvényhívás során megadott értékek***
formális paraméterek	aktuális paraméterek
formális argumentumok	aktuális argumentumok
paraméterek	argumentumok

A könyvünkben az ANSI szabvány által javasolt ***paraméterek*** és ***argumentumok*** elnevezést használjuk.

A függvényhívás során a vezérlés a hívó függvénytől átkerül az aktivizált függvényhez. Az argumentumok (amennyiben vannak) szintén átadódnak a meghívott függvénynek. A már bemutatott **return** utasítás végrehajtásakor vagy a függvény fizikai végének elérésekor a meghívott függvény visszatér a hívás helyére, és a **return** utasításban szereplő kifejezés mint függvényérték

(visszatérési érték) jelenik meg. A visszatérési érték nem más, mint a függvényhívás kifejezés értéke.

3.8.1. Függvények definíciója

A saját függvényeinket mindig definiálni kell. A ***definíció***, amelyet csak egyszer lehet megadni, a C programon belül bárhol elhelyezkedhet. Amennyiben a függvény definíciója megelőzi a felhasználás (hívás) helyét akkor, ez egyben a függvény deklarációja is.

A ***deklaráció***, amelyet a függvényhívás előtt kell elhelyeznünk a programban, a függvény nevét és visszatérési értékének típusát tartalmazza. Ez a magyarázata annak, hogy a könyvtári függvények deklarációját tartalmazó *include* állományokat miért a forrás file elején építjük be a programunkba. Ha a függvény deklarációja tartalmaz információkat a paraméterek típusáról is, akkor az ANSI C-ben bevezetett ***prototípussal*** van dolgunk. Valamely deklaráció vagy prototípusa többször is szerepelhet a programban, azonban mindegyik példánynak azonosnak kell lennie.

Nézzük meg a függvény definiálás általános formáját. Az eredeti C nyelven a függvény paramétereinek felsorolása és deklarációja elkülönült egymástól:

```
⟨a visszatérési érték típusa⟩ függvénynév (⟨paraméterlista⟩)
⟨a paraméterek deklarációja⟩;
{
   /* a függvény törzse */
   ⟨lokális definíciók és deklarációk⟩
   ⟨utasítások⟩
}
```

Az ANSI C-ben a *paraméterlista*, amelyben az egyes paramétereket vessző választja el, a paraméterek neve előtt annak típusát is tartalmazza:

```
⟨a visszatérési típus⟩ függvénynév (⟨paraméter-dekl. lista⟩)
{
   /* a függvény törzse */
   ⟨lokális definíciók és deklarációk⟩
   ⟨utasítások⟩
}
```


A ⟨ ⟩ jelek között szereplő részek hiányozhatnak a definícióból. Példaként készítsük el a régi (*oldexp*) és az új (*newexp*) definícióját annak a függvénynek, amely nem negatív egész számok egész kitevőre történő hatványozását végzi el:

```
int oldexp(alap, exp)
   int alap;
   int exp;
{
   int hv = 1;

   if (exp >0)
      for ( ; exp; exp--) hv*=alap;
   return hv;
}


int newexp( int alap, int exp )
{
   int hv = 1;

   if (exp >0)
      for ( ; exp; exp--) hv*=alap;
   return hv;
}
```

A következőkben részletesen ismertetjük a függvény deklarációban szereplő részeket, a használatukkal összefüggő szabályokat.

3.8.1.1. A függvények tárolási osztálya

A függvények definiálásakor a függvény visszatérési típusa előtt megadhatunk tárolási osztályt is. (A tárolási osztályok részletes ismertetését a következő fejezet tartalmazza.) Az alapértelmezés szerinti tárolási osztály (függvények esetén) az **extern**, amely azt jelöli, hogy a függvény más modulból is elérhető. Amennyiben a függvény elérhetőségét az adott modulra kívánjuk korlátozni, a **static** tárolási osztályt kell megadnunk. (A paraméterek deklarációjában csak a **register** tárolási osztály specifikálható.):

```
static double fv()
{
}
```

3.8.1.2. A függvények visszatérési típusa

A visszatérési típus meghatározza a függvényérték típusát, amely tetszőleges skalár (**char**, **short**, **int**, **long**, **float**, **double**, **signed**, **unsigned**, felsorolási és mutató) vagy strukturált (**struct**, **union**) típus lehet.
A függvény a **return** utasítás feldolgozásakor ad vissza értéket, amelyet (ha szükséges) a visszatérési típusra konvertál. A visszaadott érték az utasításban szereplő kifejezés értéke:

```
return kifejezés;
```

Ha a függvény definíciójában nem adjuk meg a visszatérési típust, akkor alapértelmezés szerint **int** típusú lesz a függvényérték.

A függvényen belül tetszőleges számú **return** utasítás elhelyezhető. Az alábbi faktoriálist számító függvényekben egy, illetve két **return** utasítást használunk. (A függvények önmagukat hívó, rekurzív függvények, amelyekkel külön alfejezetben foglalkozunk.)

```
int fact1(int n)
{
  return (n>1) ? n*fact1(n-1) : 1;
}

int fact2(int n)
{
   if (n>1)
      return n * fact2(n-1);
   else
      return 1;
}
```

A **void** típus felhasználásával olyan függvényeket készíthetünk, amelyek nem adnak vissza értéket. (Más programozási nyelveken ezeket az alprogramokat eljárásnak nevezzük.) Ebben az esetben a függvényből való visszatérésre a **return** utasítás kifejezés nélküli alakját használjuk:

```
return ;
```

A **void** függvényekben gyakran a függvényt törzsét záró kapcsos zárójelet használjuk visszatérésre. Az alábbi *sorminta* függvény a megadott karaktert adott számszor kiírja egymás mellé:


```
void sorminta(int db, char ch)
{
  int i;
  for (i=0; i<db; i++)
   printf("%c", ch);
}
```

3.8.1.3. Függvényattribútumok használata

A különböző C változatokban a visszatérési típus után különböző függvényattribútumok is szerepelhetnek, amelyek a függvény alapértelmezés szerinti működését módosítják. A Turbo C 2.0 rendszerben az alábbi attribútumok (módosítók) használhatók:

pascal	- a Pascal függvényhívási konvenciók használatát írja elő,
cdecl	- a C függvényhívási konvenciók használatát írja elő,
interrupt	- megszakításkezelő függvényt definiál,
near	- közeli függvényhívás (assembly függvényhez),
far	- távoli függvényhívás (assembly függvényhez),
huge	- távoli függvényhívás, az adatszegmens átadásával (assembly függvényhez).

Például a korlátozott elérhetőségű megszakítás rutin definíciója:

```
void interrupt itrutin()
{
. . .
}
```

Az alábbi **long** típussal visszatérő függvényt, a Pascal nyelvben használatos hívási konvenciókkal, távoli függvényként kell meghívni:

```
long far pascal pasfv()
{
. . .
}
```

3.8.1.4. A függvények paraméterei

A paraméterek megadásának módjában és ezzel kapcsolatosan a prototípus használatában jelentkezik a különbség a hagyományos és az ANSI függvény definíció között. A hagyományos megoldásban a paraméterlista csak a paraméterek nevét tartalmazza, míg a deklarációjukat a függvény törzse elé kell elhelyezni:

```
int oldexp(alap, exp)
   int alap;
   int exp;
{ . . . }
```

vagy

```
int oldexp(alap, exp) int alap; int exp;
{ . . . }
```

Ebben az esetben nem szükséges, hogy a paraméterek deklarációjának sorrendje megegyezzen a paraméterlistában megadott sorrenddel, sőt az azonos típusú paraméterek deklarációja - a változóknál megismert módon - összevonható:

```
int oldexp(alap, exp)
   int exp;
   int alap;
{ . . . }
```

vagy

```
int oldexp(alap, exp)
   int alap, exp;
{ . . . }
```

Amennyiben elhagyjuk a paraméterek deklarációját, akkor a fordító minden paramétert **int** típusúként használ. Így a fentiekkel egyenértékű az alábbi definíció:

```
int oldexp(alap, exp)
{ . . . }
```

Az új stílusú függvény-definíciót használva, a paraméterlistában minden paraméter előtt ott áll a paraméter típusa. Ebben az esetben értelemszerűen a

paraméterek deklarációs sorrendje követi a paraméterek sorrendjét és semmilyen összevonás nem lehetséges.

```
int newexp( int alap, int exp )
{ . . . }
```

A deklarált paraméterek a függvényen belül mint a függvény lokális változói használhatók, azonban a függvényen kívülről nem érhetők el. A paraméterek típusa az alap-, a struktúra, az unió, a mutató és a tömb típusok közül kerülhet ki.

3.8.2. Függvények deklarációja és prototípusa

A függvény d e k l a r á c i ó j a , amely általában megelőzi a függvény definícióját, meghatározza a függvény nevét és visszatérési típusát (továbbá - amennyiben megadjuk - a tárolási osztályt és a függvény attribútumait). A deklaráció nem tartalmaz információt a paraméterekről:

```
⟨a visszatérési érték típusa⟩ függvénynév();
```

A függvény deklarációját mindig pontosvesszővel kell lezárni. A bevezető példánknál maradva, a két hatványfüggvénynek megegyezik a deklarációja:

```
int oldexp();

int newexp();
```

Ha csak a függvény deklarációját használjuk, akkor a deklarált függvény tetszőleges számú és típusú argumentummal meghívható. Ekkor az argumentumok típusát a függvényhívásnál alkalmazott szabványos konverziók határozzák meg. Ezek szerint a **float** típus automatikusan **double** típussá, a **char** és **short** típusok - **int** típusúvá, míg az **unsigned char** és **unsigned short** típusok - **unsigned int** típusúvá konvertálódnak. A fentiektől eltérő típusú argumentumok megtartják a típusukat. Amennyiben deklaráció nem szerepel a függvényhivatkozás helyéig, akkor még a visszatérési típus is az alapértelmezés szerinti **int** típus lesz. A deklaráció elsősorban a függvény visszatérési típusának definiálására szolgál.

Ezzel szemben az új stílusú függvényekhez készített p r o t o t í p u s , amely a paraméterek információival bővíti deklarációt, teljes leírását tartalmazza a

függvénynek. Nézzük meg, milyen mechanizmusok érvényesülnek a program fordításakor, ha prototípust használunk:

- A prototípus definiálja a függvény visszatérési típusát, amennyiben az eltér az **int** típustól. (Azonban **int** típus esetén is ajánlott a prototípus megadása.)
- Az argumentumok konverziója a prototípusban definiált típusoknak megfelelően, nem pedig a szabványos lépések szerint megy végbe.
- A paraméterlista és az argumentumlista összevetésével a fordító ellenőrzi a paraméterek számának és típusainak összeférhetőségét.
- A prototípus felhasználható a függvénymutató inicializálására is.

(A prototípus is tartalmazhat tárolási osztály és függvényattribútum előírásokat.) A prototípus gyakorlatilag megegyezik az új stílusú függvény definíció első sorával (a függvény fejlécével), amelyet pontosvessző zár le. A prototípus általában csak a paraméterek típusát tartalmazza, amennyiben a paraméterek nevét is megadjuk, akkor azokat a fordító figyelmen kívül hagyja:

```
⟨a visszatérési érték típusa⟩ függvénynév (⟨típuslista⟩);
```

Az alábbi két prototípus megegyezik:

```
int newexp( int, int);

int newexp( int masalap, int masexp );
```

A prototípus szempontjából még két speciális formára ki kell térnünk. Azon függvények prototípusa, amelyek nem rendelkeznek paraméterrel, a

```
típus fv(void);
```

és nem a

```
típus fv();
```

hiszen ez nem más, mint a függvény deklarációja. A C nyelv lehetővé teszi, hogy a legalább egy paramétert tartalmazó paraméterlistát a `, ...` deklaráció zárja a prototípusban. Az így definiált függvény legalább egy, de különben tetszőleges számú és típusú argumentummal meghívható. Példaként tekintsük a már sokszor használt ***printf*** függvény prototípusát:

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


Amennyiben a függvény definíciója megelőzi a függvényhívásokat a programban, akkor a régi típusú függvény definíció deklaráció is egyben, míg az új stílusú függvénymegadás a prototípusnak felel meg.

Ahhoz, hogy programjainkban a függvények hívása korrekt módon menjen végbe, mindig meg kell adnunk a függvények prototípusát. A régi függvény definíciós formát csak a kompatibilitás miatt hagyta meg a szabvány és például a rokon C++ nyelvben már csak az új forma használható.

3.8.3. A függvényhívás

A függvényhívás olyan kifejezés, amely átadja a vezérlést és az argumentumokat (amennyiben vannak) az aktivizált függvénynek. A függvényhívás általános alakja:

```
kifejezés1 (⟨kifejezés2⟩)
```

ahol a *kifejezés1* a függvény neve (vagy a függvény címét szolgáltató kifejezés), míg az opcionálisan megadható *kifejezés2* az argumentum kifejezések vesszővel tagolt listája.

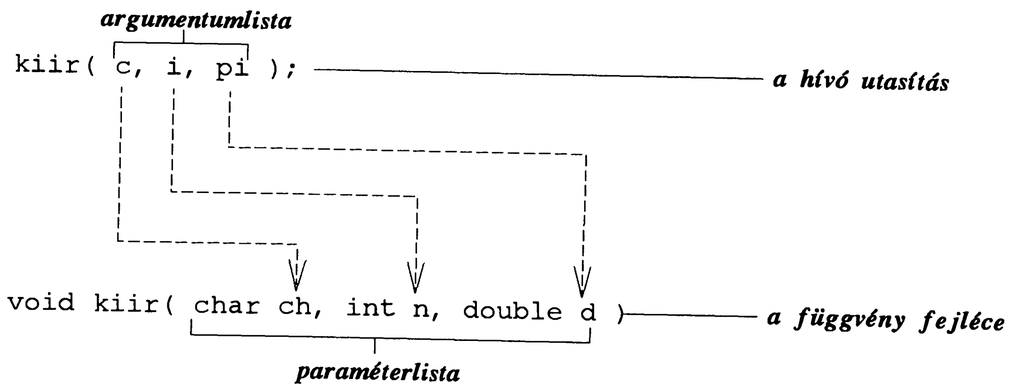
```
int fv( int a, float b );    /* prototípus */

x = fv( 4, 5.67 );           /* függvényhívás */
```

Az argumentumok kiértékelésének sorrendjét nem definiálja a C nyelv. Egyetlen dolgot garantál mindössze a függvényhívás operátora, hogy mire a vezérlés átadódik a hívott függvénynek, az argumentumlista teljes kiértékelése (a mellékhatásokkal együtt) végbemegy. Azon függvények esetén, ahol a prototípusban a paraméterlista helyén **void** kulcsszó szerepel, a híváskor nem adható meg egyetlen argumentum sem:

```
int fv(void);       /* prototípus */

x = fv();           /* függvényhívás */
```

A következő oldalon látható példában (FUNC1.C) a *kiir* függvény egyszerűen *csak* megjeleníti a hívási argumentumok értékét. A 3.25. ábrán azt ábrázoltuk, hogyan jön létre a kapcsolat az argumentumok és a paraméterek között. Az argumentumok értéke sorra átadódik a megfelelő paraméternek.

```
#include <stdio.h>

void kiir(char, int, double); /* A függvény prototípusa */

main()
{
  char   c = 'A';
  int    i = 123;
  double pi = 3.14159265;

  printf("A hívás elõtt:\t\tc=%c i=%d pi=%lf\n", c, i, pi);
  kiir(c, i, pi);                          /* a függvényhívás */
  printf("A hívás után :\t\tc=%c i=%d pi=%lf\n", c, i, pi);
}

/* A kiir függvény definíciója */
void kiir(char ch, int n, double d)
{
  printf("A függvényen belül:\tc=%c i=%d pi=%lf\n",ch,n,d);
  ch++;
   n++;
   d--;
  printf("A függvényen belül:\tc=%c i=%d pi=%lf\n",ch,n,d);
}
```



3 25 ábra
Az argumentumok és a paraméterek kapcsolata

A függvényhívásnál használt argumentumok típusa az alap-, a struktúra, az unió és a mutató típusok közül kerülhet ki. A C nyelvben az argumentumok érték szerint adódnak át a hívott függvénynek. Ez azt jelenti, hogy az

argumentum másolatát veszi fel a megfelelő paraméter értékként. Ennek következtében, ha függvényen belül a paraméteren valamilyen műveletet végzünk, annak nincs kihatása a híváskor megadott argumentumra. A FUNC1.C program futási eredményén jól látható az érték szerinti argumentum átadás mechanizmusa:

```
A hívás előtt:          c=A i=123 pi=3.141593
A függvényen belül:     c=A i=123 pi=3.141593
A függvényen belül:     c=B i=124 pi=2.141593
A hívás után :          c=A i=123 pi=3.141593
```

Hiába változtattuk meg a paraméterek értékét, a függvényből visszatérve a megváltoztatott értékek elvesztek.

Ezek után felmerül a kérdés, hogyan lehet C-ben olyan függvényt írni, amely felcseréli két egész típusú változó értékét? Az érték szerint argumentum átadás látszólag ezt nem teszi lehetővé. Ha azonban az átadott érték valamely objektumnak a címe (például pointer változó értéke), akkor a cím felhasználásával lehetőség nyílik arra, hogy a függvényből "kihivatkozva" indirekt módon megváltoztassuk az objektum értékét. Nézzük példaként az egész változók értékének felcserélését megvalósító programot (CSERE.C):

```
#include <stdio.h>
void cserel(int *, int *);       /* a prototípus */

main()
{
  int x=7, y=30;

  printf("A hívás elõtt:\t x=%d  y=%d\n", x, y);
  cserel( &x, &y );              /* a függvényhívás */
  printf("A hívás után :\t x=%d  y=%d\n", x, y);
}

/* A csere függvény definíciója */
void cserel ( int * p, int *q )
{
   int sv;

   sv = *p;
   *p = *q;
   *q = sv;
}
```

Ebben az esetben a *cserel* függvény argumentumai nem a változók értéke, hanem a változók címe (&x és &y), amit egészre mutató pointerekbe, mint paraméterekbe, veszünk át (**int** *p és **int** *q). A cserét ezek után a **p* és a **q* objektumok között végezzük el, egy *sv* segédváltozó bevezetésével.

Az elmondottak alapján módosítsuk úgy a FUNC1.C programot, hogy a függvényen belüli módosítások visszahassanak a hívó függvény változóira. A megoldás az alábbiakban látható (FUNC2.C):

```
#include <stdio.h>

void kiir(char*, int*, double*);   /* A prototípus */

main()
{
  char   c = 'A';
  int    i = 123;
  double pi = 3.14159265;

  printf("A hívás elõtt:\t\tc=%c i=%d pi=%lf\n", c, i, pi);
  kiir(&c, &i, &pi);                /* a függvényhívás */
  printf("A hívás után :\t\tc=%c i=%d pi=%lf\n", c, i, pi);
}

/* A kiir függvény definíciója */
void kiir(char * ch, int * n, double * d)
{
  printf("A függvényen belül:\tc=%c i=%d pi=%lf\n",
                                          *ch, *n, *d);
  (*ch)++;
   (*n)++;
   (*d)--;
  printf("A függvényen belül:\tc=%c i=%d pi=%lf\n",
                                          *ch, *n, *d);
}
```

Újabb olyan területre tévedtünk, ahol mutatókat kell használnunk. Más nyelveken, ahol a fordító valósítja meg a cím (referencia) szerinti argumentum átadást, kisebb szerepet kap a mutató típus. A dolog további érdekessége az, hogy az ott használt mechanizmus megegyezik azzal a megoldással, amit C nyelven a mutatók segítségével megvalósítottunk.

A fejezet összefoglalásaképpen nézzünk meg két érdekes példát. Az elsőben **void** függvény segítségével valósítsuk meg az egész típusú operandussal rendelkező "címe" operátort (&) (CIMEOP.C). A megoldásban az okoz

nehézséget, hogy mutatóra mutató pointert (**) kell ahhoz átadnunk a *cime* függvénynek, hogy a mutató a függvényen belül értéket kapjon. Másrészt az egész változó címének átadásáról (*) is gondoskodnunk kell:

```
void cime(int ** pp, int * p)
{
  *pp = p;
}

main()
{
  int   a=4730;
  int *ap;

  cime( &ap , &a );  /* Mint az ap = &a */
  *ap += 13;
  /* Az a értéke 4743 lesz! */

 }
```

A következőben olyan függvényt mutatunk be, amely mutató típusú visszatérési értékkel rendelkezik (LVFUNC.C). A függvény az argumentumként átadott címet egyszerűen visszaadja függvényértékként:

```
int * kozvetit( int * p)
{
   return p;
}

main()
{
  int a=4730;

  *kozvetit(&a) += 13;  /* Mint az a += 13 */
  /* Az a értéke 4743 lesz! */

 }
```

A függvényekkel kapcsolatos fogalmak és alapmegoldások ismertetése után a programozás során felmerülő bonyolultabb problémák megoldására mutatunk példákat. Megnézzük, hogyan lehet függvénynek függvényt, tömböt, sztringet és struktúrát átadni argumentumként. Ezt követően megismerkedünk a ***main*** függvény paraméterezési lehetőségeivel, a rekurzív függvényekkel és a változó hosszúságú argumentumlista feldolgozásával.

3.8.4. A függvény mint függvény argumentum

A programozás során általában arra törekszünk, hogy az általunk kidolgozott függvényeket, a kódjuk megváltoztatása nélkül, hatékonyan lehessen alkalmazni. Ez a cél a függvények megfelelő paraméterezésével elérhető. A matematikai alkalmazások készítése során jogos igény, hogy egy jól megvalósított (beprogramozott) algoritmust különböző függvények esetén tudjunk felhasználni. Ehhez a szükséges függvényt mint argumentumot kell átadni az algoritmust megvalósító függvénynek, amire azonban a C nyelven nincs lehetőség. Mi a teendő? A választ valószínűleg már sejti az Olvasó - mutatót kell használnunk.

3.8.4.1. A függvénytípus és a typedef

Mielőtt megismerkednénk a függvényre mutató pointerekkel, annak érdekében, hogy a deklarációink olvashatók legyenek, nézzük meg a **typedef** tárolási osztály felhasználását függvények esetén. A **typedef** segítségével a függvény típusát egyetlen szinonim névvel jelölhetjük. A függvénytípus deklarálja azt függvényt, amely az adott számú és típusú paraméterhalmazzal rendelkezik és a megadott adattípussal tér vissza.

Tekintsük például a faktoriálist számító függvényt, melynek prototípusa és definíciója:

```
unsigned long fakt(int);          /* prototípus */

unsigned long fakt(int n)         /* definíció */
{
   unsigned long f = 1;

   for ( ; n > 0 ; n--) f *= n;
   return f;
}
```

Most pedig vegyük a függvény definíció fejlécét, tegyük elé a **typedef** kulcsszót, majd pontosvesszővel zárjuk le azt. A keletkező új típus neve pedig legyen *faktfv*:

```
typedef unsigned long faktfv(int n);
```


A **typedef** deklarációban, ellentétben a prototípussal, a paraméterek nevét is érdemes megadni, mivel ekkor a típusnév a függvény definíciójában is alkalmazható. A *faktfv* típus felhasználásával a *fakt* függvény prototípusa és definíciója az alábbi alakban írható:

```
faktfv fakt; ;                    /* prototípus */

faktfv fakt                      /* definíció */
{
   unsigned long f = 1;

   for ( ; n > 0 ; n--) f *= n;
   return f;
}
```

3.8.4.2. Függvényre mutató pointerek

C nyelvben a függvénynevek kétféle módon használhatók. A függvénynevet a függvényhívás operátor baloldali operandusaként megadva függvényhívás kifejezést kapunk

```
fakt(7)
```

melynek értéke a függvény által visszaadott érték. Ha azonban a függvénynevet önállóan használjuk

```
fakt
```

akkor egy mutatóhoz jutunk, melynek értéke az a memóriacím, ahol a függvény kódja elhelyezkedik (kódpointer), típusa pedig a függvény típusa.

Definiáljunk egy olyan mutatót, amellyel a *fakt* függvényre mutathatunk, vagyis értékként felveheti a *fakt* függvény címét! A definíciót egyszerűen megkapjuk, ha a *fakt* függvény fejlécében szereplő nevet a `(*fptr)` kifejezésre cseréljük ki:

```
unsigned long (*fptr) (int);
```

Az *fptr* olyan pointer, amely **unsigned long** visszatérési értékkel és egy **int** típusú paraméterrel rendelkező függvényre mutathat.

A definíció azonban sokkal olvashatóbb formában is megadható, ha használjuk a **typedef** segítségével előállított *faktfv* típust:

```
faktfv *fptr;
```

Az *fptr* nem csak mutathat, hanem rá is mutat az alábbi értékadás végrehajtása után a *fakt* függvényre:

```
fptr = fakt;
```

Ezek után a *fakt* függvény az *fptr* mutató felhasználásával indirekt módon is meghívható:

```
f10 = (*fptr) (10);     vagy     f10 = fptr (10);
```

A **ftpr* kifejezést azért kell zárójelben használnunk, mivel a függvényhívás operátora erősebb precedenciájú az indirekt hivatkozás operátoránál. Természetesen az *fptr* mutató tetszőleges, a *fakt* függvénnyel megegyező típusú függvény címét felveheti.

A függvény neve és a függvényre mutató pointer között hasonló összefüggés van, mint a tömb neve és a tömbelem típusára mutató pointer között. (Mind tömbnév, mind pedig a függvénynév konstans mutatóként viselkedik.) Csak emlékeztetőül a tömb és a mutatók közötti kapcsolat:

```
int a[10], *p = a;

a[0] = a[1];            p[0] = p[1];,
*(a+0) = *(a+1);        *(a+0) = *(a+1);
```

Nézzük meg mi a helyzet az *fv* függvény és a rá hivatkozó *pfv* esetén:

```
void fv (int);
void (*pfv) (int) = fv;

/* A lehetséges függvényhívások: */
fv(2);                  pfv(2);
(*fv)(2);               (*pfv)(2);
```

A következő példaprogramokban szereplő *tabloz* függvény tetszőleges **double** típusú paraméterrel és **double** visszatérési értékkel rendelkező függvény értékeinek táblázatos megjelenítésére alkalmas. A *tabloz* függvény paraméterei között szerepel még az intervallum két határa és a lépésköz. A

program első változata (TABLA1.C) **typedef** nélkül listázza a négyzetre emelés és a gyökvonás függvény értékeit, míg a második változat (TABLA2.C) típusdeklaráció felhasználásával teszi ugyanezt. A TABLA1.C program forráslistája:

```
#include <stdio.h>
#include <conio.h>

/* A tablóz függvény prototípusa */
void tabloz(double (*)(double), double, double, double);

double sqr(double);  /* a saját függvény prototípusa */
double sqrt(double); /* a könyvtári függvény prototípusa */

main()
{
   printf("\n\nAz sqr függvény értékei ([-2,2] dx=0.5)\n");
   tabloz(sqr, -2, 2, 0.5);
   getch();
   printf("\n\nAz sqrt függvény értékei ([0,2] dx=0.2)\n");
   tabloz(sqrt, 0, 2, 0.2);
   getch();
}

/* A tablóz függvény definíciója */
void tabloz(double (*fp)(double), double a, double b,
            double lepes)
{
   double x;

   for (x=a; x<=b; x+=lepes)
       printf("%13.5f \t %15.10f\n", x, (*fp)(x));
}

/* Az sqr függvény definíciója */
double sqr(double x)
{
   return x * x;
}
```

A TABLA2.C program forráskódja a *matfv* függvénytípus bevezetésével sokkal áttekinthetőbb:

```
#include <stdio.h>
#include <conio.h>
```

```
/* A matematikai függvények típusa */
typedef double matfv(double x);

/* A tablóz függvény prototípusa */
void tabloz(matfv *, double, double, double);
matfv sqr;        /* a saját függvény prototípusa */
matfv sqrt;       /* a könyvtári függvény prototípusa */

main()
{
   printf("\n\nAz sqr függvény értékei ([-2,2] dx=0.5)\n");
   tabloz(sqr, -2, 2, 0.5);
   getch();
   printf("\n\nAz sqrt függvény értékei ([0,2] dx=0.2)\n");
   tabloz(sqrt, 0, 2, 0.2);
   getch();
}

/* A tablóz függvény definíciója */
void tabloz(matfv *fp, double a, double b, double lepes)
{
  double x;
  for (x=a; x<=b; x+=lepes)
      printf("%13.5f \t %15.10f\n", x, (*fp)(x));
}

/* Az sqr függvény definíciója */
matfv sqr
{
  return x * x;
}
```

3.8.4.3. Függvényre mutató pointerek tömbje

C nyelven gyakran használt, hasznos adatstruktúra a függvényre mutató pointerek tömbje, amely lehet egy- vagy többdimenziós. Elsősorban olyan esetekben használjuk ezeket a tömböket, amikor valamilyen választást (pl. menüből menüpont kiválasztása) gyorsan kívánunk feldolgozni.

Tegyük fel, hogy a programunk vezérlését egyszerű menüvel kívánjuk megoldani:

```
1. Beolvas
2. Számol
3. Kiír
----------
0. Kilép
```


A hagyományos megoldás során ciklusban fogadjuk a karaktereket, és az **if** vagy a **switch** utasítással kiválasztjuk a megfelelő esetet (MENU1.C):

```
#include <stdio.h>
#include <conio.h>

/* A menü definíciója */
char * menu[] = { "\n", "1. Beolvas",
                        "2. Számol",
                        "3. Kiír",
                        "----------",
                        "0. Kilép" , "\n", NULL };
/* Globális változók, közös használatra */
int a, b, c;
/* A függvények prototípusa */
void beolvas(void);
void szamol(void);
void kiir(void);

main()
{
  char ch , **p;

  do {
      /* A menü kiírása */
      p = menu;
      while (*p)
          printf("%s\n", *p++);
      /* A választás feldolgozása*/
      ch = getch();
      switch (ch) {
         case '0':
            break;
         case '1':
            beolvas();
            break;
         case '2':
            szamol();
            break;
         case '3':
            kiir();
            break;
         default:
            putchar('\a');
      }
  } while (ch != '0');
}
```

```
/* A menüpontnak megfelelő függvények definíciója */
void beolvas(void)
{
  printf("Kérek két számot [a,b]: ");
  scanf("%d,%d",&a,&b);
}

void szamol(void)
{
  c = a + b;
}

void kiir(void)
{
  printf("%d + %d = %d\n", a, b, c);
}
```

Nézzük meg, hogyan módosul a ***main*** függvény mutatótömb bevezetésével! A fenti három (azonos típusú) függvény címének tárolására használható *menupont* vektor definíciója (kezdőértékek megadásával) az alábbiakban látható:

```
void (*menupont[3]) (void) = { beolvas, szamol, kiir };
```

A ***main*** függvény sokkal rövidebb és áttekinthetőbb lett. Meg kell jegyeznünk, hogy minél több a menüpont (például kétdimenziós menü esetén), annál hatékonyabb ez a megoldás. A tömbelem által kijelölt függvény (például a *szamol*) meghívása:

```
(* menupont[1])();    vagy    menupont[1]();
```

A módosított ***main*** függvény:

```
main()
{ char ch , **p;
  void (*menupont[3]) (void) = { beolvas, szamol, kiir };
  do {
      p = menu;
      while (*p)  printf("%s\n", *p++);
      ch = getch();
      if (ch > '0' && ch < '4')
          menupont[ch-'1'] ();
      else if (ch != '0')
              putchar('\a');
  } while  (ch!='0');
}
```


Hasonlóan érdekes probléma az egyszerű, kötött szerkezetű leírónyelven megírt programok (például NC, HP plotter, stb.) gyors értelmezése. A parancsok nagy száma miatt a vezérlő függvény áttekinthetetlen és kusza lesz, ha a szokásos **if** vagy **switch** szerkezetet használjuk. Szép megoldáshoz jutunk olyan mutatótömb kialakításával, melynek elemei a parancsoknak megfelelő függvényekre mutatnak.

Tegyük fel, hogy a bemutatásra kerülő program által feldolgozott leírónyelv utasításai "**An** *paraméterek*" szerkezetűek. Az első betű ('A'-'Z') a parancs csoportját, míg a második számjegykarakter ('0'-'9') a parancsot jelöli ki. Ez összesen 26x10=260 különböző parancs megadását teszi lehetővé. A kétkarakteres parancsot szóközzel elválasztva követik a parancs paraméterei.

A feldolgozó függvények egy mutatót kapnak argumentumként, amely az utasításban a *paraméterek* rész elejére mutat. Ezért ezen függvények típusa:

```
void parancs(char *);
```

A 260 parancsot kezelő függvény címének tárolására kétdimenziós tömböt használunk, így a parancs azonosítását az utasítás első két karaktere alapján elvégezhetjük:

```
void (* fvt[26][10]) (char *);
```

A **typedef** segítségével megírt program az FVTOMB.C állományban található, míg az FVTOMB0.C file a **typedef** nélküli megoldást tartalmazza, amely az alábbiakban tanulmányozható:

```
#include <stdio.h>

void p0(char *);
void p1(char *);
void q0(char *);
void ures(char *);

/* A feldolgozandó programrészlet */
char *prog[]={ "P0 100,300",
               "P1 Hello",
               "P2 100,LEFT",
               "P0 200,500",
               "Q1 500",
               "P1 Bye",
               "Q0 "};
```

```
main()
{
   void (* fvt[26][10]) (char *);
   int i, j, n;

   /* Inicializálás:  ures */
  for (i = 0; i < 26; i++ )
      for (j = 0; j < 10; j++ )
          fvt[i][j] = ures;

   /* Inicializálás:  p0, p1 és q0 */
  fvt['P'-'A']['0'-'0'] = p0;
  fvt['P'-'A']['1'-'0'] = p1;
  fvt['Q'-'A']['0'-'0'] = q0;

  /* A példaprogram feldolgozása */
  n=sizeof(prog)/sizeof(prog[0]);
  for ( i = 0; i < n; i++ )
   (*fvt[ prog[i][0]-'A' ][ prog[i][1]-'0' ])(&prog[i][3]);
}

/* A parancsfüggvények definíciója */
void ures (char * p)
{
  printf("**** Érvénytelen parancs ****\n");
}

void p0 (char * p)
{
  int a,b;
  sscanf(p, "%d,%d", &a, &b);
  printf("%d + %d = %d\n", a, b, a+b);
}

void p1 (char * p)
{
  printf("%s\n", p);
}

void q0 (char * p)
{
  printf("--- A feldolgozás vége ---\n");
}
```

Ugyancsak érdemes egy pillantást vetni a feldolgozás eredményeként megjelenő képernyőtartalomra:


```
100 + 300 = 400
Hello
**** Érvénytelen parancs ****
200 + 500 = 700
**** Érvénytelen parancs ****
Bye
--- A feldolgozás vége ---
```

3.8.5. Struktúra átadása függvénynek

A struktúrák (**struct** és **union**) argumentumként történő felhasználása semmi gondot nem jelent. Az ANSI C nyelven választhatunk az érték szerinti és a mutató segítségével történő megoldások között. (Az eredeti C nyelv csak ez utóbbit tette lehetővé.) Az ANSI C nyelvben a struktúra típus függvény visszatérési értéke is lehet.

A két lehetőség közül általában ki kell választanunk az adott feladathoz legjobban illeszkedő megoldást. A választásnál szempontként felhasználható, hogy a memóriaigény és a futási idő szempontjából egyaránt a mutatót használó megoldás a hatékonyabb.

Példaként vegyük a komplex számok tárolására alkalmas struktúrát:

```
struct cplx {
       double re, im;
       };
```

Készítsünk a **struct** *cplx* típushoz szinonim típusnevet a **typedef** felhasználásával:

```
typedef struct cplx complex;
```

A függvények definíciójában a **struct** *cplx* és a *complex* típusneveket egyaránt felhasználhatjuk. Először készítsünk függvényt két komplex szám összeadására (*sum1*). Az összeadandó komplex számokat és az eredmény tárolására szolgáló struktúrát a mutatójuk segítségével adjuk át a függvénynek:

```
void sum1 (complex *pa, complex *pb, complex *pc)
{
  pc->re = pa->re + pb->re;
  pc->im = pa->im + pb->im;
}
```

A függvényen belül a *complex* struktúra adattagjaira a nyíl operátor felhasználásával hivatkozunk.

A második példánkban (*sum2*) az összeg kiszámításához szükséges komplex számokat érték szerint adjuk át, hiszen ezeket nem kívánjuk megváltoztatni Figyeljünk oda, hogy az adattagok eléréséhez a megfelelő operátort használjuk:

```
void sum2 (complex pa, complex pb, complex *pc)
{
  pc->re = pa.re + pb.re;
  pc->im = pa.im + pb.im;
}
```

A *sum3* függvény visszatérési értékként szolgáltatja a két érték szerint átadott komplex szám összegét. Az összegzést egy lokális struktúrában végezzük el, melynek értékét a **return** utasítással adjuk vissza. (A függvény fejlécében a teljes típusneveket használtuk):

```
struct cplx sum3 (struct cplx a, struct cplx b)
{
  complex c;
  c.re = a.re + b.re;
  c.im = a.im + b.im;
  return c;
}
```

A három különböző megoldáshoz három különböző hívási mód tartozik. Az alábbi programrészlet mind a három összegző függvény meghívását tartalmazza:

```
main()
{
  struct cplx c1 = {4, 7}, c2 = {26, 13}, c3;

  sum1(&c1, &c2, &c3); /*Mindhárom argumentum pointer    */

  c3.re = c3.im =0;    /*A c3 nullázása                  */
  sum2(c1, c2, &c3);   /*Két struktúra és egy mutató arg.*/

  c3.re = c3.im =0;    /*A c3 nullázása                  */
  c3 = sum3(c1,c2);    /*Két struktúra argumentum        */
}
```


3.8.6. Tömb argumentumok használata

Tömbökkel végzett műveletek gyakran szerepelnek a C programokban. Amennyiben ezeket a műveleteket általánosítani szeretnénk, függvénybe kell foglalnunk az elvégzendő tevékenységsorozatot. Most megnézzük, hogy milyen lehetőségeket biztosít a C nyelv tömbök függvénynek való átadására.

Már a legelején le kell szögeznünk, hogy tömb argumentumot nem lehet érték szerint (a teljes tömb átmásolásával) függvénynek átadni. Sőt különbség van az egydimenziós (vektorok) és a többdimenziós tömbök argumentumként való átadása között. A fejezetben külön tárgyaljuk a vektorok és a kétdimenziós tömbök esetét.

3.8.6.1. Vektor argumentumok

Egydimenziós tömbök (vektorok) függvény argumentumként való megadása esetén a tömb első elemére mutató pointer adódik át. Más szavakkal a `T[]` típusú vektor argumentum `T*` típusú mutatóként adódik át a hívott függvénynek. Ebből a megoldásból viszont az is következik, hogy a vektor elemein a függvényen belül végrehajtott változtatások a függvényből való visszatérés után is megmaradnak.

A vektorok átadásának fenti módszere elegendő ahhoz, hogy a vektor elemeit elérjük (gondoljunk a vektor és a mutatók közötti analógiára), azonban semmilyen információ nem jut el a függvényhez a vektor méretével (elemeinek számával) kapcsolatban. Ezért ezt az adatot egy második argumentum felhasználásával szoktuk megadni. Kivételt képeznek azok az egydimenziós karaktertömbök, amelyekben sztringet tárolunk, hisz sztringek esetén a memória tartalmazza a sztring végét jelölő 0-ás byte-ot.

Nézzünk néhány jellegzetes megoldást a vektorokat feldolgozó függvények kialakítására. Az alábbi példák mindegyike az átadott egész elemű vektor elemeinek átlagát határozza meg és adja vissza függvényértékként. A hívás bemutatásához használjuk az alábbi 10 elemű vektort:

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

Az első példában `int*` mutatóként fogadjuk a vektor átadott kezdőcímét és a függvény törzsében is pointer műveletekkel érjük el az elemeket:

```
double atlag1 (int * vektor, int n)
{
  long sum = 0;
  int i;

  for (i = 0; i < n; i++)
      sum += *(vektor+i);
  return (double)sum / n;
}
```

A következő példában `int[]` típussal deklaráljuk a függvény *vektor* paraméterét. Az `int[]` deklaráció csak annyit közöl a vektorról, hogy egész elemeket tartalmaz, így hatása megegyezik az `int*` deklarációéval. (Ha a zárójelek között valamilyen egész kifejezést adunk meg, akkor azt figyelmen kívül hagyja a fordító.)

```
double atlag2 (int vektor[], int n)
{
  long sum = 0;
  int i;

  for (i = 0; i < n; i++)
      sum += vektor[i];
  return (double)sum / n;
}
```

Mindkét függvényt ugyanúgy kell meghívni:

```
b1 = atlag1 (a, 10);

b2 = atlag2 (a, 10);
```

A vektor címét és méretét más módon is megadhatjuk, ha ezt a két adatot egy struktúrában egyesítjük:

```
typedef struct {
          int *ptr;      /* a vektor címe   */
          int size;      /* az elemek száma */
          } vec;
```

Ezt a struktúrát értékként adjuk át az átlagot számító függvénynek, melynek a definíciója:


```
double atlag3 (vec vs)
{
  long sum = 0;
  int i;

  for (i=0; i < vs.size; i++)
      sum += vs.ptr[i];
  return (double)sum / vs.size;
}
```

A meghívás módja természetesen eltér az előző függvényekétől, hisz a hívás előtt fel kell töltenünk egy *vec* típusú struktúrát a szükséges adatokkal:

```
vec v;

v.ptr  = a;
v.size = 10;
b2     = atlag3 (v);
```

A kétdimenziós tömbök esetén körülményesebben kell eljárnunk. Azonban gyakran mutatókat tartalmazó vektort használunk a kétdimenziós tömbök helyett. Az ilyen vektor argumentumként való átadása az előzőekben bemutatott megoldások valamelyikével elvégezhető, mint ahogy az a következő példából is látható. A *wsprint* függvény az átadott sztring tömb (karakterre mutató pointerek vektora) elemeit egymás mellé kiírja. A sztringtömbben a sztringeket ***NULL*** pointer zárja.

A megoldást, mind a vektoros (*wsprint2*), mind a pointeres (*wsprint1*) felfogásban az alábbi program tartalmazza. (Felhívjuk a figyelmet arra, hogy a példában a vektor elemeinek típusa `char*`.)

```
#include <stdio.h>

char *Desc[] ={ "Cogito", "ergo", "sum.", NULL };

void wsprint1(char **p)
{
  while (*p)
        printf("%s ",*p++);
  printf("\n");
}
```

```
void wsprint2(char *p[])
{
  int i=0;
  while (p[i])
        printf("%s ",p[i++]);
  printf("\n");
}

main()
{
  wsprint1(Desc);
  wsprint2(Desc);
}
```

3.8.6.2. Kétdimenziós tömb argumentumok

A kétdimenziós tömb elnevezés alatt most csak a fordító által létrehozott tömböket értjük:

```
int a[3][4];
```

A tömb elemeire való hivatkozás (`a[i][j]`) mindig átírható a `*((int*)a+(i*4)+j)` formula szerint (ezt teszik a fordítók is). Ebből a kifejezésből az is látszik, hogy a kétdimenziós tömb második dimenziója (*4*) alapvető fontossággal bír a fordító számára, míg a sorok száma tetszőleges lehet.

A feladat az, hogy készítsünk olyan függvényt, amely tetszőleges méretű kétdimenziós tömb elemeit megjeleníti mátrixos formában. Első lépésként azonban írjuk meg a függvénynek azt a változatát, amely csak 3x4-es tömbök megjelenítésére alkalmas:

```
                                    /* hívás: print_mx34(a); */
void print_mx34(int mx[3][4])
{
 int i,j;

 for (i=0; i<3; i++) {
    for (j=0; j<4; j++)
      printf("%d\t",mx[i][j]);
    printf("\n");
}
```


A kétdimenziós tömb is a terület kezdőcímét kijelölő mutatóként adódik át a függvénynek. Azonban a fordító az elemek elérése során:

```
*( (int *)mx + (i*4)+j )
```

figyelembe veszi azt, hogy a sorok 4 elemet tartalmaznak. Ezért a fenti függvény egyszerűen átalakítható olyan függvénnyé, amely nx4-es tömb kiírására alkalmas, csak a sorok számát kell átadni második paraméterként:

```
                                    /* hívás: print_mxn4(a, 3); */
void print_mxn4(int mx[][4], int n)
{
 int i,j;

 for (i=0; i<n; i++) {
    for (j=0; j<4; j++)
      printf("%d\t",mx[i][j]);
    printf("\n");
}
```

Arra azonban nincs lehetőség a C nyelvben, hogy a második dimenziót is elhagyjuk, hiszen akkor a fordító nem képes a tömb sorait azonosítani. Egyetlen dolgot tehetünk annak érdekében, hogy általános megoldást tudjunk megvalósítani, ha átvesszük a tömb területének elérését a fordítótól (a fenti kifejezés felhasználásával):

```
                          /* hívás: print_mxnm((int *)a, 3, 4); */
void print_mxnm(int *mx, int n, int m)
{
 int i,j;

 for (i=0; i<n; i++) {
    for (j=0; j<m; j++)
      printf("%d\t", mx[i*m+j] ) ;
    printf("\n");
 }
```

3.8.6.3. Sztring argumentumok

A sztring argumentumok az egydimenziós karaktertömböknek megfelelően adódnak át a függvényeknek, amelyek átadását már tisztáztuk. Az ok, amiért mégis külön alfejezetet szentelünk e témának az, hogy a sztringek feldolgozásánál használt fogásokat bemutassuk.

A sztringek kezelése során alapvetően két megközelítési módot használhatunk. Az első esetben, mint vektort kezeljük a sztringet (indexek alapján), a másik megközelítés szerint mutató segítségével végezzük el a szükséges műveleteket. Az első példában mindkét megoldást bemutatjuk az ***strcpy*** (sztring másolás) könyvtári függvény működését megvalósító függvényben. A további példákban azonban felváltva használjuk a két megközelítési módot.

Az ***strcpy*** függvény

```
char * strcpy (char * celstr, char * forrasstr);
```

a *forrasstr* sztring tartalmát átmásolja a *celstr* sztringbe és a célsztringre mutató pointerrel tér vissza. A működést mutatókkal megvalósító függvény:

```
char * pstrcpy(char *p, char *q)
{
   char *s = p;  /* A célterület */

   while (*p++ = *q++);
   return s;
}
```

A *pstrcpy* függvényben a *p* és *q* mutatókkal dolgozunk. Ahhoz, hogy a célterületet kijelölő mutatót a függvény végén vissza tudjuk adni, az *s* segédváltozóban megőrizzük azt. A függvénynek nincs külön paramétere, amely a forrás sztring hosszát tartalmazná. Erre nincs is szükségünk, hisz a sztringet záró '\0' karakter (0-ás byte) vizsgálatával végig tudunk lépkedni a sztringen. A sztringeken való végighaladást (++) és a karakterek másolását (=) egyetlen **while** ciklusba sűrítettük. A ciklus akkor áll le, amikor a 0-ás byte is átmásolódott. A ciklust természetesen kevésbé tömör formában is fel lehet írni:

```
while (*p = *q) {
      p++;
      q++;
}
```

vagy

```
while (*q) {
     *p = *q;
      p++;
      q++;
}
*p=0;
```


Amennyiben vektorként kívánjuk a sztringet feldolgozni, szükségünk van egy index változóra, amellyel a vektorokat indexeljük:

```
char * vstrcpy(char p[], char q[])
{
   int i;         /* Indexeléshez kell */
   char *s = p;   /* A célterület */

   i = 0;
   while (p[i] = q[i]) i++;
   return s;
}
```

A másolást leállító feltétel szintén a sztringet záró 0-ás byte elérése. Ehhez a byte-hoz az index változó (*i*) léptetésével jutunk el. Összehasonlítás kedvéért írjuk fel a másoló ciklus kevésbé tömör alakját:

```
for (i=0 ; q[i]; i++)
   p[i] = q[i];
p[i] = 0;
```

Láthatjuk, hogy mutatók segítségével tömörebben lehet megfogalmazni a feladat megoldását. Azonban ez a tömörség a program olvashatóságát lényegesen rontja.

A következő függvény az első argumentumként átadott sztringhez hozzámásolja a másik argumentumban megadott sztring tartalmát (***strcat***):

```
char * pstrcat(char *p, char *q)
{
 char *s = p;

   while (*p) p++;        /* A célsztring végére lépkedés */
   while (*p++ = *q++);   /* Másolás */
   return s;
}
```

A *vindex* függvény az *s1* sztringben megkeresi az *s2* sztring első előfordulását, és visszatér a helyet azonosító indexszel. A -1 függvényérték azt jelzi, hogy nem található meg az *s2* az *s1*-ben:

```
int vindex(char s1[], char s2[])
{
   int i, j, k;

   for (i=0; s1[i]; i++)    /* Lépkedés az s1-ben */
   {
      /* Az s1 i. pozíciójától van-e az s2 ? */
      for (j=i, k=0; s2[k] && s1[j] == s2[k]; j++, k++);

      /* Ha a leállás feltétele az s2 vége - benne van! */
      if (s2[k] == '\0') return i;
   }
  return -1;                /* Nincs benne */
}
```

Az utolsó példánkban szereplő *strequ* függvény összetett feladatot lát el. A függvény feladata megállapítani, hogy két sztring (*s* és *t*) egyforma-e vagy sem. Amennyiben a két sztring tartalma megegyezik, akkor a függvény értéke 1, különben pedig 0 lesz. Ezen kívül a függvény beállít két mutatót (**ps* és **pt*) a sztringek azon karakterére, ahol az eltérés jelentkezett. (Egyezőség esetén a sztringet záró 0-ás byte-ra mutatnak a pointerek.) A megoldást a megfelelő hívási móddal együtt az STREQU.C program tartalmazza, melynek listája:

```
#include <stdio.h>
#include <string.h>

/* A függvény prototípusa */
int strequ(char *, char *, char **ps, char **pt);

main()
{
    char s1[32], s2[32];
    char *p1, *p2;

    /* Azonos sztringek esetén */
    strcpy(s1,"C nyelv");
    strcpy(s2, s1);

    if (strequ(s1, s2, &p1, &p2))
       printf("A két sztring azonos!\n");
    else
       printf("A két sztring eltérõ részei: %s   %s \n",
              p1, p2);
```



```
    /* Különbözõ sztringek esetén */
    strcpy(s1, "Hello mindenki!");
    strcpy(s2, "Hello mindenhol!");

    if (strequ(s1, s2, &p1, &p2))
       printf("A két sztring azonos!\n");
    else
       printf("A két sztring eltérõ részei: %s   %s \n",
              p1, p2);
}

/* A függvény definíciója */
int strequ(char *s, char *t, char **ps, char **pt)
{
  int equ;
  *ps = s; /* A *ps és a *pt a sztringekre mutatnak */
  *pt = t;
  /* A két sztring karakterenkénti összehasonlítása */
  /* legfeljebb az első sztring végéig              */
  while (( equ = (**ps == **pt)) && (**ps != 0))
  {
       ++*ps;    /* Léptetés a sztringekben */
       ++*pt;
  }
  return equ;
}
```

3.8.6.4. Konstans paraméterek

Az érték szerinti argumentumátadás biztosítja azt, hogy az argumentumként megadott változók értékét a függvényen belülről nem lehet megváltoztatni. Ha azonban az adott változóra mutató pointert adjuk meg argumentumként, akkor a változó értéke indirekt módon a függvényen belülről is megváltoztatható.

Mivel a tömbök és a sztringek átadása mindig mutatóval történik, ezért azok tartalma belépve a függvénybe, akár véletlenül is, megváltozhat. Amennyiben biztosítani szeretnénk, hogy a mutatott objektumok tartalma ne változzék meg, a paraméter deklarációban a **const** típusminősítőt kell használnunk.

A szabványos könyvtári függvények prototípusában gyakran előfordul ez a típusmódosító. Azokat az azonosítókat jelölik, melyek értéke nem változik meg a függvényen belül.

Példaként tekintsük a sztringek átadásánál használt kétféle paraméter deklarációt, és gyűjtsük össze az adott tfpusú függvényben a fordító által engedélyezett és tiltott műveleteket:

```
void pfv(const char * p)
{
    p++;            /* ok! */
   *p = 'A';        /* hiba! */
    p[0] = 'B';     /* hiba! */
}

void vfv(const char p[])
{
    p++;            /* hiba! */
   *p = 'A';        /* hiba! */
    p[0] = 'B';     /* hiba! */
}
```

Azonban a C nyelvben a **const** előírás betartása csak a közvetlen hivatkozásokra terjed ki. Ez azt jelenti, hogy a **const** mutató értékét egy nem konstans pointernek átadva, a fenti műveletek minden további nélkül elvégezhetők:

```
char *q = (char *)p;
q++;
*q = 'A';
q[0] = 'B';
```

Ennek ellenére a **const** szó használata javasolt azon sztringek átadásánál, amelyeket nem kívánunk a függvényen belül módosítani. Ez a megoldás, mint ahogy az alábbi példában is látható, bizonyos nehezen kideríthető programhibák kiszűrését segítheti. Nézzük először a sztringek hosszának meghatározására használható függvény helyes alakját:

```
int strlen(const char * p)
{
  char *s = (char *)p;
  while (*p++);
  return p-s-1;
}
```

Ha a **const** típusminősítőt nem adjuk meg, és a **while** feltételében felcseréljük az operátorok sorrendjét (például rosszul emlékeztünk a könyvben elmondottakra), akkor olyan függvényhez jutunk, amely mindig -*1*-et ad vissza, és az átadott sztringet törli (az első karakterét nullázva):


```
int strlen(char * p)                    /*  ! hibás ! */
{
  char *s = (char *)p;
  while (++*p);
  return p-s-1;
}
```

A **const** kulcsszó megadása után a fordító azonnal jelzi a hibát a ++*p kifejezésnél, amely nem a mutatót, hanem a mutatott karaktert lépteti.

3.8.7. A main függvény paraméterei és visszatérési értéke

A C programon belül a ***main*** függvénynek kitüntetett szerepe van, hisz kijelöli a program belépési pontját. (Ezért a ***main*** függvény csak egyetlen példányban adható meg.)

A ***main*** függvény különlegességét nem csak az adja, hogy a program végrehajtása vele kezdődik, hanem az is, hogy nulla, egy vagy két paramétere lehet:

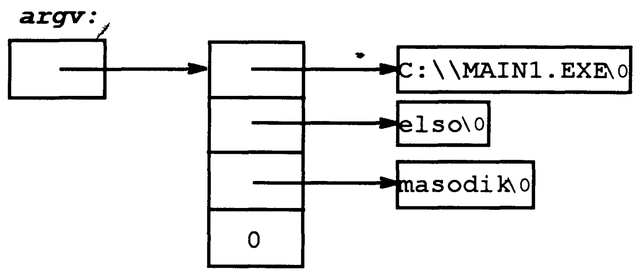
```
int main( )

int main( int argc) ( )

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

Néhány C implementáció lehetővé teszi további paraméterek használatát. Az MS-DOS alatti C fordítók többsége egy harmadik paraméter megadását támogatja:

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

A paraméterek neveit tetszőlegesen megválaszthatjuk, azonban a program olvashatóságát jelentősen javítja a szabványos elnevezések használata.

Az *argv* egy sztring mutatókat tartalmazó tömbre (vektorra) mutat (3.26. ábra), az *argc* pedig a tömbben található sztringek számát adja meg. (Az *argc* értéke legalább 1, mivel az *argv[0]* mindig a program nevét tartalmazó sztringre mutat.)



3 26 ábra
Az argv paraméter értelmezése

A ***main*** visszatérési értékét, amely általában **int** típusú, a ***main*** függvényen belüli **return** utasításban, vagy a program tetszőleges pontján az ***exit*** könyvtári függvény argumentumában adhatjuk meg. Az STDLIB.H állomány szabványos konstansokat is tartalmaz,

```
#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1
```

amelyeket kilépési kódként használva, a program sikeres, illetve sikertelen futását jelölhetjük. A ***main*** függvényt **void** típusúnak definiálva a programunknak határozatlan lesz a kilépési kódja.

A ***main*** argumentumainak megadása, illetve a visszatérési érték feldolgozása függ az operációs rendszertől. A könyvünkben csak az MS-DOS operációs rendszer alatti megoldásokra hagyatkozunk. MS-DOS-ban az argumentumokat a programot indító parancssorból másolja be a futtató rendszer az ***argv*** által kijelölt tömbbe.

Ha a program neve MAIN1, akkor a a hívási argumentumok megadására a

```
C:\TC\MAIN1 elso masodik 3. 4.
```

parancssor használható. Az argumentumok elválasztására a szóközt használjuk.

Az alábbi MAIN1.C program kiírja a parancssor argumentumok számát, megjeleníti az argumentum sztringeket és az *envp* által mutatott sztringtömb tartalmát. Az *envp* DOS környezeti változókat tartalmazó sztringtömböt jelöl ki, melyben az érvényes elemeket egy ***NULL*** pointer zárja.


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

int main(int argc, char *argv[], char **envp)
{
  int i;
  char **p;

  printf("Az argumentumok száma: %d\n\n",argc);
  for (i=0; i<argc; i++)
      printf("%d. argumentum: \"%s\"\n",i,argv[i]);

  p = envp;
  while (*p)
      printf("DOS környezeti változó: \"%s\"\n",*p++);
  return (EXIT_SUCCESS);
}
```

A program futásának eredménye, a fenti parancssort használva:

```
Az argumentumok száma: 5

0. argumentum: "C:\MAIN1.EXE"
1. argumentum: "elso"
2. argumentum: "masodik"
3. argumentum: "3."
4. argumentum: "4."

DOS környezeti változó: "COMSPEC=C:\COMMAND.COM"
DOS környezeti változó: "PATH=C:\DOS;C:\DOS\HELP;
DOS környezeti változó: "PROMPT=$P$G"
```

Az alábbi MAIN2.C program a segédprogramok írásánál jól felhasználható megoldást mutat be az indítási argumentumok helyes megadásának tesztelésére. A program csak akkor indul el, ha pontosan két argumentummal indítjuk. Ellenkező esetben hibajelzést és a program indításának formáját bemutató üzenetet ír ki a képernyőre:

```
Hibás paraméterezés!

A program indítása: MAIN2 arg1 arg2
```

A MAIN2:C program szövege a következő oldalon tekinthető meg.

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

int main(int argc, char *argv[])
{
  int i;

  if (argc !=3 ) {
      printf("\n\aHibás paraméterezés!\n");
      printf("\nA program indítása: MAIN2 arg1 arg2\n");
      return EXIT_FAILURE;
  }

  printf("\nHelyes paraméterezés:\n");
  printf("1. argumentum: %s\n", argv[1]);
  printf("2. argumentum: %s\n", argv[2]);

  return EXIT_SUCCESS;
}
```

A DOS alatt futó programok visszatérési értékét (kilépési kódját) a *batch* file (.BAT) ***ERRORLEVEL*** változójával lehet lekérdezni.

3.8.8. Rekurzív függvények használata

A matematikában lehetőség van bizonyos adatok és műveletek rekurzív definiálására. Minden r e k u r z í v problémának létezik i t e r a t í v (ciklust használó) megoldása, amely általában sokkal nehezebben programozható, de hatékonysága miatt mégsem szabad megfeledkezni róla! Klasszikus példaként tekintsük először a *Fibonacci* néven is ismert *Leonardo de Pisa* nyúl feladatát:

> "Hány nyúlpárunk lesz 3,4,5,...,n hónap múlva, ha egy nyúlpár kéthónapos kortól kezdve havonta egy-egy új párt hoz világra, feltéve, hogy az új párok is ezen törvény alapján szaporodnak, és mind életben maradnak."

A megoldást a Fibonacci számok sora tartalmazza (ha a 0 kezdőelemet figyelmen kívül hagyjuk):

```
0, 1, 1, 2, 3, 5, 8, 13, 21, ...
```


A sor n-dik elemének meghatározására az alábbi rekurziós szabály szolgál:

$$a_0 = 0$$
$$a_1 = 1$$
$$a_n = a_{n-1} + a_{n-2}, \qquad n: 2, 3, 4, \ldots$$

A rekurziós szabály alapján elegáns megoldást kapunk, ha önmagát hívó rekurzív függvényt használunk. Ekkor gyakorlatilag a fenti összefüggést fogalmazzuk át C programmá:

```
unsigned long fibr( int n )
{
   if (n<2)
      return n;
   else
      return fibr(n-1) + fibr(n-2);
}
```

A rekurzív megoldás általában rövidebb és áttekinthetőbb, mint az iteratív megoldás, azonban a számítási idő és a memóriaigény jelentős növekedése miatt az esetek többségében mégis az iteratív megoldás használatát javasoljuk:

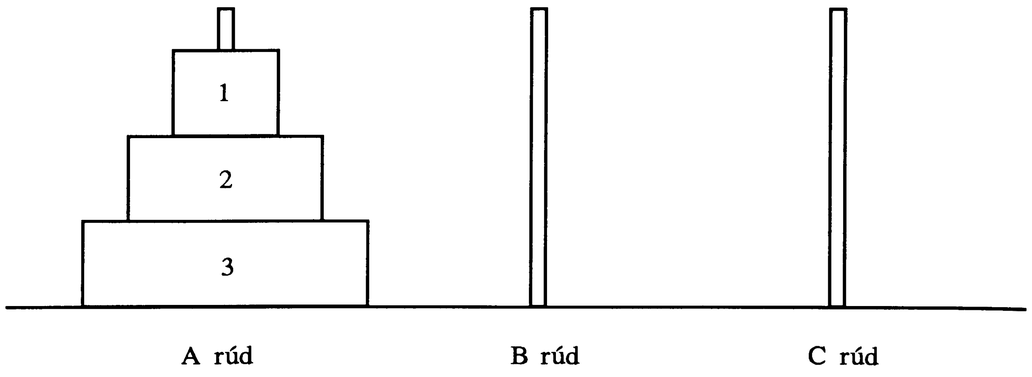
```
unsigned long fib( int n )
{
  unsigned long f0 = 0, f1 = 1, f2 = n;

  while (n-- > 1) {
       f2 = f0 + f1;
       f0 = f1;
       f1 = f2;
  }
  return f2;
}
```

A következő feladat megoldása a rekurzió alkalmazása nélkül igen bonyolult. A megoldandó probléma a Hanoi tornyai nevet viseli.

Adott három rúd: *A*, *B* és *C*. Az *A* rúdon induláskor *N* darab különböző átmérőjű lyukas korong helyezkedik el, nagyságuk szerinti csökkenő sorrendben (lásd. 3.26.) ábra.

A feladat a korongok átrakása a *C* rúdra, az alábbi szabályok figyelembe vételével:

- a *B* rúd közbenső tárolásra használható,
- minden lépésben csak egy korong mozgatható,
- minden korong csak nálánál nagyobb átmérőjű korongra helyezhető.



3.27 ábra
Az Hanoi tornyai játék kiindulási helyzete

(A feladat egy legendán alapul, amely szerint Hanoi közelében található kolostorban 64 aranykorongból álló tornyot raknak át a szerzetesek, a fenti szabályok betartásával, minden nap egyetlen korongot mozgatva. A legenda szerint akkor lesz vége a világnak, ha az átrakást befejezik.)

A feladat megoldását az alábbi HANOI.C példaprogram tartalmazza:

```
#include <stdio.h>

void Honnan_Hova_Mozgatas(char, char);
void Hanoi(int, char, char, char);

main()
{
  int  korongszam;

  printf("\nA korongok száma : ");
  scanf("%d", &korongszam);
  printf("\n");
  Hanoi(korongszam, 'A', 'B', 'C');
}
```



```
void Hanoi(int n, char honnan, char mivel, char hova)
{
    if (n == 1)
       Honnan_Hova_Mozgatas(honnan, hova);
    else {
       Hanoi(n-1, honnan, hova, mivel);
       Honnan_Hova_Mozgatas(honnan, hova);
       Hanoi(n-1, mivel, honnan, hova);
    }
}   /* Hanoi */

void Honnan_Hova_Mozgatas(char innen, char ide)
{
    printf("Tedd a korongot a(z)\t%3c\t", innen);
    printf(" rúdról a(z) \t%3c\t rúdra.\n", ide);
}  /* Honnan_Hova_Mozgatas */
```

Három korong megadása esetén a program által javasolt megoldás:

```
A korongok száma : 3

Tedd a korongot a(z)     A     rúdról a(z)     C     rúdra.
Tedd a korongot a(z)     A     rúdról a(z)     B     rúdra.
Tedd a korongot a(z)     C     rúdról a(z)     B     rúdra.
Tedd a korongot a(z)     A     rúdról a(z)     C     rúdra.
Tedd a korongot a(z)     B     rúdról a(z)     A     rúdra.
Tedd a korongot a(z)     B     rúdról a(z)     C     rúdra.
Tedd a korongot a(z)     A     rúdról a(z)     C     rúdra.
```

3.8.8.1. A rekurzív alprogramok csoportosítása

A rekurzív algoritmusok általában önrekurzióra vagy kölcsönös rekurzióra épülnek.

Önrekurzió:

Önrekurzióról akkor beszélünk, amikor egy függvény közvetlenül hívja önmagát. A fenti két példában ezt a megoldást használtuk. A rekurzív működés megértéséhez tekintsük a faktoriális számítását, amely rekurzív megoldással szintén hosszasan működik, így ebben az esetben is az iteratív (ciklusos) megoldás használata javasolt. A faktoriális számítás rekurzív definíciója:

$$n! = \begin{cases} 1, & \text{ha } n=0 \\ n*(n-1)!, & \text{ha } n>0 \end{cases}$$

Ennek alapján, a `4!` kiszámításának lépései az alábbi szemléletes formában ábrázolhatók:

```
4! = 4 * 3!
     3 * 2!
         2 * 1!
             1 * 0!
                 1  = 4 * 3 * 2 * 1 * 1 = 24
```

A fenti számítási menetet megvalósító C függvény:

```
unsigned long factr (int n)
{
    if (n == 0)
       return 1;
     else
       return n * factr (n-1);
}
```

A faktoriálissal összefüggő, rekurzióval megoldható probléma egy szó betűinek összes lehetséges módon történő összerakása (ismétlés nélküli permutáció). A megoldáshoz használt három függvény (***main***, *swapnm* és *permut*) közös globális változókat használ. A PERMUT.C állomány tartalmazza az alábbiakban megtekinthető programot.

```
#include <stdio.h>
#include <string.h>

/*Globális változók a függvények közötti kummunikációhoz*/
char szo[8];
int  nbetu, nperm;
void swapnm(char, char);
void permut(int);

main() {
   printf("\nKérek egy max. 7 betûs szót: ");
   gets(szo);
   nbetu = strlen(szo);
   nperm = 1;
   if (nbetu > 0 && nbetu <8)
      permut(nbetu-1);
}
```



```
/* A permutációt elõállító rekurzív függvény */
void permut(int k)
{
   int i;
   for (i=0; i<=k; i++) {
       swapnm(i, k);
       if (k)
          permut(k-1);
       else
          printf("\n%3d - %s",nperm++, szo);
      swapnm(k, i);
   }
}

/* Két karakter felcserélése a szóban */
void swapnm( char n, char m)
{
    char z;

    z = szo[n];
    szo[n] = szo[m];
    szo[m] = z;
}
```

A program a működését során sorszámozza az előállított permutációkat:

```
Kérek egy max. 7 betûs szót: KAT

  1 - ATK
  2 - TAK
  3 - TKA
  4 - KTA
  5 - AKT
  6 - KAT
```

Kölcsönös rekurzió

Kölcsönös rekurzióról akkor beszélünk, amikor egy függvény az általa meghívott másik függvényből hívódik meg újra. Kölcsönös rekurzió esetén különösen oda kell figyelni a rekurziót leállító feltétel megadására. Az ilyen megoldásokban ajánlott a függvények prototípusának megadása:

```
typedef void fv(int);          /* A függvények típusa */
fv fv_a, fv_b;                 /* A függvények prototípusa */
```

```
void fv_a(int i)
{
  . . .
  if (i > 0) fv_b(i / 2);    /* Az fv_b hívása */
  . . .
}

void fv_b(int j)
{
   . . .
   if (j > 0) fv_a(j - 1);  /* Az fv_a hívása */
   . . .
}
```

3.8.9. Változó hosszúságú argumentumlista

Bizonyos függvények esetén nem lehet pontosan megadni az argumentumok számát és típusát. Az ilyen függvények deklarációjában a paraméterlistát három pont zárja:

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

A három pont azt jelenti a fordítóprogram számára, hogy "még lehetnek további argumentumok". A ***printf*** esetén legalább egy argumentumnak kell szerepelnie, amelyet tetszőleges további argumentum követhet:

```
printf("Hello C !\n");
printf("A nevem: %s \n", nev);
printf("Az összeg: %d + %d = %d\n", a, b, c);
```

Felvetődik a kérdés, honnan tudja a ***printf*** vagy a ***scanf***, hogy hány argumentumot kell feldolgoznia? A választ a formátumsztring adja, melyen végiglépkedve a formátum alapján dolgozza fel a ***printf*** a soron következő argumentumot.

Mivel az ilyen deklarációjú függvények hívásakor a fordító csak a "..." listaelemig képes az argumentumok típusát egyeztetni, ezért a további argumentumokra a hagyományos konverziókat hajtja végre. Más szavakkal, ebben az esetben a megadott argumentumok (esetleg konvertált) típusa szerint megy végbe az argumentumok átadása a függvénynek.

A C nyelv lehetővé teszi, hogy saját függvényeinkben is használjuk a három pontot - az ún. változó hosszúságú argumentumlistát. Ahhoz, hogy a

paramétereket tartalmazó memóriaterületen megtaláljuk az átadott argumentumok értékét, legalább az első paramétert mindig meg kell adnunk. Az argumentumok feldolgozására két megoldás közül választhatunk. Az első esetben, ismerve az adott implementáció memóriahasználatát, mutatók segítségével olvassuk fel az argumentumok értékét. A másik megoldás, amely a változó hosszúságú argumentumlista esetén is biztosítja a C programok hordozhatóságát, az STDARG.H *include* file-ban definiált makrok alkalmazását jelenti.

3.8.9.1. MS-DOS specifikus feldolgozás

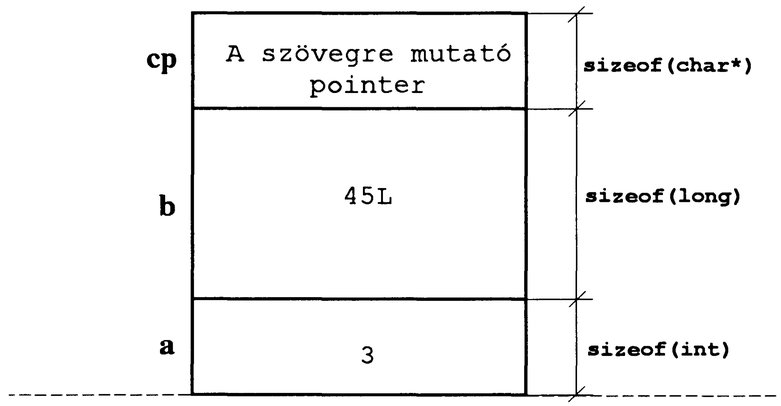
Az MS-DOS alatt fejlesztett C programok esetén ismernünk kell az argumentumok fizikai átadási módját. A paraméterek és a lokális objektumok tárolására a verem (*stack*) nevű memóriaterületet használja a Turbo C fordító. Amikor egy

```
int fv(int a, long b, char * cp);
```

prototípusú függvényt meghívunk

```
fv(3, 45L, "Hello");
```

akkor az argumentumok egymás után a verembe másolódnak a 3.27. ábrán ábrázolt módon. Amennyiben ismerjük az első paraméter helyét (*&a*) és ismerjük a további paraméterek méretét, akkor ezek alapján minden argumentumot elérhetünk a *b* és *cp* paraméternevek nélkül is.



3 28 ábra
A stack szerkezete argumentumok átadásakor

Amennyiben a *b* és a *cp* nem szerepelnek a paraméterlistában

```
int fv(int a, ... );
```

az elmondottak alapján fel tudjuk dolgozni az utolsó két argumentumot.

A VA_ARG1.C példaprogramban a számok összegzését és átlagolását végző függvények változó hosszúságú argumentumlistát használnak. Az *osszeg* függvény esetében az egész argumentumok listáját 0-val zárjuk, míg az *atlag* függvénynél az átadott első egész argumentum tartalmazza a további **double** típusú argumentumok darabszámát.

```
#include <stdio.h>

/*Az <stdarg.h> csak a VA_ARG2.C programban szükséges!*/
#include <stdarg.h>

/* A függvények prototípusa */
int osszeg(int, ...);
double atlag (int, ...);

main()
{
   int s;
   double a;

   /* A függvények meghívása */
   s = osszeg(1, 2, 3, 4, 5, 0);

   /* Az első arg. egész, a többi pedig double */
   a = atlag(4, 10.20, 20.30, 30.40, 40.50);
}

 /* A függvények definíciója */
int osszeg(int elso, ...)
{
  int *p = &elso; /* Az elso nevű paraméterre áll! */
  int  s = elso;

  while (* ++p)
        s+= *p;
  return s;
}
```



```
double atlag(int n, ...)
{
  int    *p = &n+1; /* Az n utáni paraméterre áll! */
  double *q = (double *)p;
  double  s = 0;
  int     i;

  for (i=0; i<n; i++)
       s += *q++;
  return s / n;
}
```

Felhívjuk a figyelmet arra, hogy a fenti függvények meghívásakor különös gondot kell fordítanunk arra, hogy a megadott argumentumok olyan típusúak legyenek, mint amilyet a függvényben várunk.

3.8.9.2. Szabványos feldolgozás

A C szabvány tartalmaz néhány olyan rutint, amelyek segítségével a változó hosszúságú argumentumlista feldolgozásához nem kell ismerni az adott számítógépes környezet "lelkivilágát". Az STDARG.H *include* állományban deklarált, illetve definiált rutinok a következők:

va_arg	az argumentumlista következő elemét adja vissza,
va_end	"nagytakarítás" az argumentumok feldolgozása után,
va_start	inicializálja az argumentumok eléréséhez használt mutatót.

A fenti rutinok előfordító makrokat jelölnek, és mindhárman a ***va_list*** típusú mutatót használják az argumentumok eléréséhez. (A szabvány javaslata alapján csak a ***va_arg*** és a ***va_start*** rutinok makrok, míg a ***va_end*** rutint könyvtári függvényként kell megvalósítani.)

Az előző VA_ARG1.C programot gépfüggetlenné tettük azáltal, hogy szabványos úton dolgozzuk fel a változó hosszúságú argumentumlista elemeit. Az alábbiakban csak a megváltozott *osszeg* és *atlag* függvényeket közöljük, a teljes programot pedig a lemezmellékleten található VA_ARG2.C file tartalmazza:

```
/* Egész számok összegzése 0-ig */
int osszeg(int elso, ...)
{
    va_list  ap;
    int      s = elso, ertek;

    va_start( ap, elso);    /* Az első argumentum átlépése */

    while (ertek = va_arg(ap, int)) /* Az int argumentumok */
          s+ = ertek;

  va_end(ap);
  return s;
}

/* Adott számú double érték átlagolása */
double atlag(int n, ...)
{
  va_list  ap;
  double   s = 0;
  int      i;

  va_start(ap, int);      /* Az első argumentum átlépése */

  for (i=0; i<n; i++)
      s += va_arg(ap,double);  /* A double argumentumok */

  va_end(ap);
  return s / n;
}
```

3.8.9.3. Az argumentumlista továbbadása

A szabványos C könyvtár tartalmaz néhány olyan függvényt, amelyek mutatóval kijelölt argumentumlistát várnak argumentumként. Ezeket a '*v*' betűvel kezdődő függvényeket arra használhatjuk, hogy a saját függvényünk változó hosszúságú argumentumlistáját továbbítjuk nekik. A C könyvtárban jól ismert függvények párjaként szerepelnek ezek a függvények:

printf	***vprintf***
fprintf	***vfprintf***
sprintf	***vsprintf***
scanf	***vscanf***
fscanf	***vfscanf***
sscanf	***vsscanf***

Az alábbi példában szereplő hibaüzenetet megjelenítő függvény legalább annyit tud, mint a ***printf*** könyvtári függvény (VA_ARG3.C):

```
#include <stdarg.h>
#include <stdio.h>

/* A függvény prototípusa */
void hibauzenet(char * formatum, ...);

main()
{
    hibauzenet(NULL);
    hibauzenet("Csak szöveg!\n");
    hibauzenet("Az x értéke %d\n",123);
    hibauzenet("Sérült állomány: %s\n", "ADATOK.DAT");
}

 /* A hibaüzenet függvény definíciója */
void hibauzenet(char * formatum, ...)
{
  va_list  pfmt;

  if (formatum != NULL) {  /* Ha van üzenet */
      va_start( pfmt, formatum);
      printf("\n\a***HIBA***\t");
      vprintf(formatum, pfmt);
      va_end(pfmt);
  }
}
```

A program indítása után az alábbi üzenetek jelennek meg a képernyőn:

```
***HIBA***     Csak szöveg!
***HIBA***     Az x értéke 123
***HIBA***     Sérült állomány: ADATOK.DAT
```

3.8.10. C deklarációk értelmezése és készítése

A C nyelv deklarációinak formája (szintaxisa) jelentősen különbözik más programozási nyelvekben használt deklarációktól. A pontos jelentése egy-egy bonyolultabb C deklarációnak még a gyakorlott C programozóknak sem azonnal érthető. Ezért ebben a fejezetben a C típusok összefoglalásaként kitérünk arra, hogyan kell értelmezni, illetve elkészíteni a C deklarációkat.

A C nyelv ugyanazt az operátor- és szimbólumkészletet használja a deklarációkban, mint a kifejezésekben szereplő azonosítók esetén. Például az alábbi példában egy egész *x* és egy **px* mutatót hozunk létre:

```
int x;
int *px;
```

A **px* deklarátornak ugyanaz a formája, mint amikor kifejezésben a mutatott egész objektumot jelenti:

```
x = *px;
```

A példában bemutatott szimmetria jól használható a kifejezések és a deklarátorok típusának megállapításában. A **px* olyan egész objektum, amelyre a *px* pointer mutat.

A C szabvány a deklaráció és a definíció fogalmak mellett a d e k l a r á t o r fogalmat is használja. Deklarátor alatt a deklarálandó objektumot és hozzákapcsolt módosítókat (mutató *, tömb [] vagy függvény ()) értjük. A szabvány legalább 12 szint mélységig javasolja a deklarátorok egybeépítésének biztosítását.

3.8.10.1. C deklarációk értelmezése

A fentinél bonyolultabb deklarációk értelmezéséhez azonban további magyarázatok szükségesek. Fontos megjegyeznünk azt, hogy a C deklarátorokban használt szimbólumok valójában C operátorok, amelyekre érvényes a precedencia és az asszociativitás szabálya. A precedencia szerint az alábbi két csoportból kerülnek ki a deklarátorokban használt operátorok:

1. A függvényhívás operátora (()) és a tömbindexelés operátora ([]) azonos precedenciával rendelkeznek. Ezen operátorok csoportosítása balról-jobbra történik.
2. Az egyoperandusú csillag (*) az indirekt hivatkozás operátora. Több csillag használata esetén a csoportosítás jobbról-balra történik.

Tekintsük a következő példát:

```
int *x[3];
```


A kérdés az, hogy ez most egészre mutató pointerek tömbje, vagy pedig egészeket tartalmazó tömbre mutató pointer? A "megfejtéshez" a kifejezést szétbontjuk, méghozzá a precedencia és az asszociativitás szabályokkal ellentétes irányban:

1. `*x[3]` egy egész,
2. `x[3]` egészre mutató pointer,
3. `x` egészre mutató pointerek 3 elemű tömbje.

A legbonyolultabb deklarátorok és kifejezések értelmezése gyorsan elvégezhető ezzel a szétbontásos módszerrel. (A csillagot azért vettük le először, mivel kisebb a precedenciája, mint az indexelés operátoré.)

A szétbontás általános szabályát is megfogalmazhatjuk. Először sorban levesszük a legkisebb precedenciájú operátorokat. Ezt követően, ha a megmaradt operátorok precedenciája megegyezik, akkor nézzük az asszociativitást. A balról-jobbra csoportosított operátorok esetén először a jobbszélső, míg a jobbról balra csoportosított operátorok esetén először a balszélső operátort vesszük le. Minden egyes lépés után megfogalmazzuk, hogy mi az, amit kaptunk. A lebontásnál természetesen a zárójeleket is figyelembe kell venni.

Ezzel a szabállyal könnyedén értelmezhetjük a leggyakrabban használt deklarációkat. Sajnos a magyar nyelv nem teszi lehetővé, hogy a típusok leírását az angol nyelvben használt egyszerűséggel írjuk fel. Nézzünk néhány példát:

`char **argv`	karakterre mutató pointerre mutató pointer,
`int a[5]`	egészeket tartalmazó 5 elemű tömb,
`int (*pa)[5]`	egészeket tartalmazó 5 elemű tömbre mutató pointer,
`int *pv[5]`	egészre mutató pointereket tartalmazó 5 elemű vektor,
`char *fv(int)`	függvény, amely **int** argumentumot kap és karakterre mutató pointerrel tér vissza,
`char (*pfv)(int)`	egy olyan függvényre mutató pointer, amely int argumentumot kap karakterre mutató pointerrel tér vissza,

Az Olvasó megnyugtatására közöljük, hogy ezeknél bonyolultabb deklarációk nem túl gyakran fordulnak elő a C programokban.

Tekintsük az alábbi egyszerűnek nem nevezhető deklarációkat, és értelmezzük azokat. A deklarációk értelmezéséhez a lebontás elvét használjuk.

```
int (* (*x()) [6]) ();
```

1	`(*(*x())[6])()`	egy egész,
2	`*(*x())[6]`	függvény, amely egészet ad vissza,
3.	`(*x())[6]`	egészet visszaadó függvényre mutató pointer,
4.	`*x()`	egészet visszaadó függvényre mutató pointerek 6 elemű tömbje,
5.	`x()`	mutató az egészet visszaadó függvényre mutató pointerek 6 elemű tömbjére,
6.	`x`	függvény, amely egészet visszaadó függvényre mutató pointerek 6 elemű tömbjére mutató pointerrel tér vissza.

Az előzőhöz formailag hasonló, azonban lényegileg különböző az alábbi deklaráció:

```
int *(* (*y) [6]) ();
```

1	`*(*(*y)[6])()`	egy egész,
2	`(*(*y)[6])()`	egészre mutató pointer,
3	`*(*y)[6]`	egészre mutató pointert visszaadó függvény,
4	`(*y)[6]`	egészre mutató pointert visszaadó függvényre mutató pointer,
5.	`*y`	egészre mutató pointert visszaadó függvényre mutató pointerek 6 elemű tömbje,
6.	`y`	egészre mutató pointert visszaadó függvényre mutató pointerek 6 elemű tömbjére mutató pointer.

A fenti két függvénydeklaráció után nézzünk egy tömbdeklarációt:

```
char (* (*z[2])() )[5];
```

1.	`(*(*z[2])())[5]`	egy karakter,
2.	`*(*z[2])()`	5 elemű karaktertömb,
3.	`(*z[2])()`	5 elemű karaktertömbre mutató pointer,
4.	`*z[2]`	5 elemű karaktertömbre mutató pointert visszaadó függvény,
5.	`z[2]`	5 elemű karaktertömbre mutató pointert visszaadó függvényre mutató pointer,
6.	`z`	5 elemű karaktertömbre mutató pointert visszaadó függvényre mutató pointereket tartalmazó 2 elemű tömb

3.8.10.2. C deklarációk készítése

A program írása szempontjából a saját deklarációk megfelelő kialakítására kell a hangsúlyt helyezni. A deklaráció felépítése az előző alfejezetben alkalmazott lebontási módszerrel ellentétes művelet, így az ott alkalmazott szabályokat is fordítva kell alkalmaznunk. Azonban az igazán megbízható és olvasható deklarációk csak a **typedef** típusdeklaráció segítségével készíthetők.

A példaként elkészített deklaráció elég bonyolult, azonban ennek megértése után az egyszerűbb (és az összetettebb) deklarációk írása már nem jelenthet gondot.

Készítsük el annak a függvénynek a deklarációját, amely karakterre mutató pointerek 5 elemű tömbjére mutató pointerrel tér vissza!

`f`	függvény, amely karakterre mutató pointerek 5 elemű tömbjére mutató pointerrel tér vissza,
`f()`	karakterre mutató pointerek 5 elemű tömbjére mutató pointer,
`*f()`	karakterre mutató pointerek 5 elemű tömbje,
`(*f())[5]`	karakterre mutató pointer,
`*(*f())[5]`	karakter,
`char *(*f())[5];`	a kész deklaráció!

A megoldást a **typedef** felhasználásával, a deklarációk elemi lépéseiből állítjuk elő:

```
/* karakterre mutató pointer típusa */
typedef char *cp;

/* karakterre mutató pointerek 5 elemû tömbjének típusa */
typedef cp v5cp[5];

/* karakterre mutató pointerek 5 elemû tömbjére mutató  */
/* pointer típusa                                       */
typedef v5cp *pv5cp;

/* olyan függvény típusa, amely karakterre mutató poin- */
/* terek 5 elemû tömbjére mutató pointerrel tér vissza  */
typedef pv5cp fpv5pc();
```

```
/* a g olyan függvény, amely karakterre mutató pointerek */
/* 5 elemű tömbjére mutató pointerrel tér vissza          */
pv5cp g();

/*az f olyan függvény, amely karakterre mutató pointerek */
/* 5 elemű tömbjére mutató pointerrel tér vissza          */
char *(*f())[] ;

/* A deklarációk ekvivalenciájának ellenőrzése */
fpv5pc *pg=f;
```

Ellenőrző kérdések

1. Mit nevezünk függvénynek?
2. Hasonlítsa össze a függvény definíciója, deklarációja és prototípusa fogalmakat!
3. Írja fel a függvény definíció és a függvényhívás általános formáját!
4. Mi a különbség az argumentum és a paraméter között?
5. Hogyan adjuk meg a függvény visszatérési értékét?
6. Hogyan lehet a függvénynek tömböt, sztringet és struktúrát átadni?
7. Kaphat-e a függvény valamilyen függvényt argumentumként?
8. Ismertesse a ***main*** függvény paramétereinek és visszatérési értékének felhasználását!
9. Értelmezze az önrekurzió és a kölcsönös rekurzió fogalmakat!
10. Mikor érdemes rekurziót használni?
11. Mit jelent a három pont a függvény paraméterlistájában?
12. Értelmezze az alábbi deklarációkat!

```
int (*fp1) (int *);
int (*p[5])(double *);
```

Feladatok

1. Írjon olyan C függvényt, amely a paraméterként átadott *n* pozitív számról eldönti, hogy prímszám-e. A függvény visszatérési értékei logikai IGAZ prímszám esetén, illetve logikai HAMIS, ha a szám nem prímszám. (PRIM.C)

2. Írjon egy-egy **int**, **long** és **double** típusú köbreemelő függvényt. (CUBE.C)

3. A háromszög három oldalának adatait olvassa be és ellenőrizze le a háromszög megszerkeszthetőségét (két oldal összegének mindig nagyobbnak kell lenni, mint a harmadik oldal). Írjon függvényeket a háromszög kerületének és területének kiszámítására. A terület kiszámítására használja fel a Heron-képletét! (HAROMSZ.C)

4. Definiáljon egy 50 elemű tömböt és adott elemszámra töltse fel értékekkel. Írjon beolvasó, kiíró függvényt, valamint számítsa ki az adatok átlagát, összegét, és keresse meg azokat az elemet, amelyik legjobban illetve legkevésbé tér el az átlagtól. Írjon olyan függvényt, amely megszámolja, hogy a tömbnek mennyi pozitív, negatív és zérus eleme van. (TOMB1.C)

5. Definiáljon egy 50 elemű tömböt és adott elemszámra töltse fel értékekkel. Írjon függvényt, amely megszámolja hogy mennyi páros és páratlan eleme van a tömbnek, valamint egy rendező függvényt, amely a tömb elemeit növekvő sorrendbe rendezi. (TOMB2.C)

6. Számítsa ki az alábbi sorozat összegét

```
h = 1 + 1/2 + 1/3 ... +1/n
```

   Ovassa be az *n* értékét! Oldja meg a feladatot különböző ciklus utasításokkal külön függvényekkel. (SOR.C)

7. Írjon függvényt, amely egy számot római számként ír ki! (ROMAISZ.C)

8. A grafikát ismertető fejezet áttanulmányozása után, írja meg a Hanoi torony példaprogram grafikus változatát! (GRHANOI.C)

3.9. Tárolási osztályok

Ahhoz, hogy igazán megértsük a C program működését, fontos ismernünk azokat a szabályokat, amelyek meghatározzák, hogy miként lehet használni a különböző változókat és függvényeket a programon belül. Ahhoz, hogy a C fordító korrekt kapcsolatot tudjon kialakítani az azonosítók és a memóriaobjektumok között, szükséges, hogy minden azonosító rendelkezzen legalább két jellemzővel - típussal és tárolási osztállyal.

A típus, amellyel a 3.3. fejezetben részletesen foglalkoztunk, meghatározza az objektum számára lefoglalt memóriaterület méretét, illetve a benne tárolt érték értelmezését.

A tárolási osztály egyrészt meghatározza, hogy az objektum hol jön létre a memóriában (regiszterben, statikus vagy dinamikus területen), másrészt pedig definiálja az objektum élettartamát. A tárolási osztályt (**auto**, **register**, **static**, **extern**) megadhatjuk a deklarációkban, de ha onnan hiányzik, akkor maga a fordító határozza meg azt, a deklarációnak a programszövegben való elhelyezkedése alapján. Mielőtt rátérünk a tárolási osztályok részletes tárgyalására tisztáznunk kell néhány, a témával szoros kapcsolatban álló fogalmat:

- élettartam (*lifetime*),
- láthatóság (*visibility*),
- érvényességi tartomány (*scope*),
- kapcsolódás (*linkage*),
- névterület (*name space*).

Mivel ezen fogalmak többsége kimondottan C specifikus, ezért az angol nyelvű szakirodalomban való eligazodás érdekében a fogalmak angol elnevezését is feltüntetjük.

3.9.1. Az azonosítók élettartama

Az élettartam (*lifetime*) a program végrehajtásának olyan időszaka, amelyben az adott változó vagy függvény létezik. Az élettartam és a tárolási osztály szoros kapcsolatban állnak. Az élettartam szempontjából az

azonosítókat három csoportra oszthatjuk: globális (statikus), lokális (automatikus) és dinamikus élettartamú objektumok.

Statikus (globális) élettartam

Azok az azonosítók, amelyeket a **static** vagy **extern** tárolási osztállyal rendelkeznek statikus élettartamúak. Például minden függvény és minden külső (a függvényekkel azonos) szinten definiált azonosító globális élettartamú.

A statikus élettartamú (globális) azonosító számára kijelölt memóriaterület (és a benne tárolt érték) a program futásának teljes időtartama alatt megmarad. A globális változók inicializálása egyetlen egyszer - a program indításakor - megy végbe.

Felhívjuk a figyelmet arra, hogy az azonosító élettartama és elérhetősége nem azonos fogalmak. Vannak esetek, amikor a program végrehajtásának teljes ideje alatt létező globális azonosító nem érhető el a program tetszőleges pontjáról (ilyen például a statikus lokális változó vagy a statikus globális változó.) Az azonosítók elérhetőségét az élettartam és a láthatóság együtt határozza meg.

Automatikus (lokális) élettartam

A függvényen (blokkon) belül a **static** tárolási osztály nélkül definiált azonosítók automatikus élettartammal rendelkeznek. Szintén automatikus élettartammal rendelkeznek a függvényen belül deklarált kapcsolódás nélküli azonosítók (lásd a 3.9.3. fejezetet) és a függvények paraméterei.

Az automatikus élettartamú (lokális) azonosító memóriaterülete (és a benne tárolt érték) csak abban a blokkban létezik, amelyben az azonosítót definiáltuk. A lokális azonosítóhoz a blokkba történő minden egyes belépéskor új terület kerül lefoglalásra, ami blokkból való kilépés során megszűnik (tartalma elvész). Következésképpen, ha a lokális változót kezdőértékkel látjuk el, akkor az inicializálás mindig újra megtörténik, amikor a változó létrejön.

Dinamikus élettartam

Dinamikus élettartammal rendelkeznek azok a memóriaterületek, amelyeket a felhasználó könyvtári függvények segítségével lefoglal, illetve felszabadít.

3.9.2 Érvényességi tartomány és a láthatóság

A tárolási osztállyal ugyancsak szoros kapcsolatban álló fogalom az azonosítók láthatósága (*visibility*), illetve érvényességi tartománya (*scope*). Az azonosító csak az érvényességi tartományán belül látható. Az érvényességi tartomány a program azon részét jelöli ki, amelynek határain belül az adott azonosítót felhasználhatjuk az objektum elérésére. Az érvényességi tartományok több fajtáját különböztetjük meg: blokk szintű (lokális), file szintű (globális) vagy függvény szintű. (Az ANSI szabvány a függvény prototípus szintű érvényességi tartomány fogalmát is bevezeti.)

A fenti szinteken az azonosítók láthatóságát a következő szabályok határozzák meg:

- A blokk szintű érvényességi tartománnyal rendelkező azonosító csak abban a blokkban látható, amelyikben deklaráltuk. Amikor a program eléri a blokkot záró '}' zárójelet, az azonosító többé nem lesz elérhető.

- A file szintű érvényességi tartománnyal rendelkező azonosító abban a fordítási egységben látható, amely a deklarációját tartalmazza. Csak azok az azonosítók rendelkeznek file szintű érvényességi tartománnyal, amelyeket globálisan, vagyis minden függvényen kívül deklarálunk. Amennyiben valamely függvényből olyan globális változót kívánunk használni, amelynek definícióját egy másik file tartalmazza, akkor az **extern** tárolási osztály felhasználásával kell az azonosítót deklarálni (külső azonosító). A C nyelvben a függvények mindig külső objektumok, hisz nem lehet a függvény definíciókat egymásba ágyazni.

- Az egyetlen függvény szintű érvényességi tartománnyal rendelkező C nyelvi egység az utasításcímke, amely csak a függvényen belül látható.

- Azok a paraméter azonosítók, amelyeket a függvények prototípusában megadunk (használatuk nem kötelező), csak a prototípust lezáró pontosvesszőig - a prototípus szintű érvényességi tartományban - láthatók.

Az alábbi SCOPE.C példaprogram bemutatja a különböző érvényességi tartományokat és az információrejtés elvének érvényesülését:

```
#include <stdio.h>

/* globális i */
int i = 13;

/* i a prototípus érvényességi tartományban */
void func1(int i);
void func2(void);

main(void)
{
   func1(23);
   func2();
   return 0;
}

void func1(int p)
{
   /* A lokális i elfedi a globális i-t! */
   int i = p;

   for (;;) {
     /* A blokk érvényességi tartományban definiált i   */
     /* elfedi mind a lokális, mind pedig a globális i-t*/
     int i = 33;
     printf("\t\tblokk   : i = %d\n", i);
     goto kilep;
   }

   /* A kilep címke függvény szintű érvényességi      */
   /* tartománnyal rendelkezik                        */
   kilep:

   /* A blokkban definiált i megszűnt - a lokális i: */
   printf("\tlokális : i = %d\n", i);
}
```

```
void func2(void) /
{
   /* A lokális i megszűnt - a globális i: */
    printf("globális: i = %d\n", i);
}
```

3.9.3. A kapcsolódás

Az érvényességi tartomány fogalma hasonló a kapcsolódás (*linkage*) fogalmához, azonban nem teljesen egyezik meg azzal. Az azonosító nevek különböző érvényességi tartományokban különböző azonosítókat jelölhetnek, (mint ahogy az előző példaprogramban is látható). Azonban a különböző érvényességi tartományban deklarált, illetve az azonos érvényességi tartományban egynél többször deklarált azonosító nevek a kapcsolódás mechanizmusának felhasználásával ugyanarra az objektumra vagy függvényre hivatkozhatnak. A kapcsolódás kijelöli azt a programrészt, amelyben az adott azonosítóra hivatkozhatunk (láthatóság). A szabvány három fajta kapcsolódást különböztet meg: belső, külső és amikor nincs kapcsolódás.

- A belső kapcsolódású azonosítók csak egyetlen fordítási egységen (modulban) belül ismertek. Ha a file szintű érvényességi tartománnyal rendelkező objektum- vagy függvény azonosítók deklarációja tartalmazza a **static** kulcsszót, akkor belső kapcsolódású azonosító jön létre, amely más fordítási egységből nem érhető el. (Ellenkező esetben külső kapcsolódással rendelkeznek a globális azonosítók.)
- A külső kapcsolódású azonosítók több fordítási egységben (modulban) is ismertek. Azok a globális objektum- vagy függvény azonosítók, amelyek deklarációjában nem szerepel tárolási osztály, vagy az **extern** tárolási osztály szerepel, külső kapcsolódásúak.
- A kapcsolódás nélküli azonosítók - valamely függvény vagy blokk helyi (lokális) azonosítói - nem rendelkeznek állandó memóriaterülettel. C nyelven az alábbi azonosítók nem rendelkeznek kapcsolódással:
  - minden olyan azonosító, amely nem objektumot vagy függvényt jelöl (például a struktúra adattag, a struktúranév, az **enum** konstansok, címkék, stb.),
  - a függvény paraméterek,
  - olyan blokk érvényességi tartománybeli objektum azonosítók, melynek deklarációjában nem szerepel az **extern** kulcsszó.

3.9.4. A névterületek

A fordító a programban használt neveket (azok felhasználási módjától függően) különböző területeken (névterület - *name space*) tárolja. Valamely névterületen belül tárolt neveknek egyedinek kell lenni, azonban a különböző névterületeken azonos nevek is szerepelhetnek. Két azonos név, amelyek azonos érvényességi tartományban helyezkednek el, de nincsenek azonos névterületen, különböző azonosítókat jelölnek. A C fordító az alábbi névterületeket különbözteti meg:

Utasításcímkék:

A névvel ellátott utasításcímke, amelyet mindig kettőspont ':' követ, része az utasításnak. Nem szükséges, hogy a különböző függvényekben használt címkék eltérőek legyenek.

Struktúra, unió és felsorolás nevek

Ezek a nevek a struktúra, az unió és a felsorolás típusspecifikáció részét képezik, és mindig közvetlenül a **struct**, **union** és **enum** kulcsszavak után állnak Azonos láthatósággal rendelkező struktúrák, uniók és felsorolások neveinek különbözniük kell egymástól

Struktúrák és uniók adattagjai

Az adattagok nevei az adott struktúrához lefoglalt névterületen helyezkednek el. Ezért valamely adattag nevet egyidejűleg több struktúrában és unióban is felhasználhatunk. Természetesen egy adott struktúrán (unión) belül az adattagok nevei nem egyezhetnek meg Az adattagok elérése csak a pont (.) vagy a nyíl (->) operátor megadásával lehetséges.

Közönséges nevek

Minden más név - a változók, a függvények, a paraméterek, a lokális változók és az **enum** konstansok nevei - közös névterületen tárolódnak. Az azonosító nevek egymásba ágyazott láthatósággal rendelkeznek, ami azt jelenti, hogy egy új blokkban újradefiniálhatók.

Típusnevek

A típusnevet azonos érvényességi tartományon belül nem lehet azonosító neveként felhasználni.

A következő programban (NAMESP.C) bemutatjuk a névterületek elvének érvényesülését. A programban ugyanazt a nevet (*bla*) többféle összefüggésben

használjuk. (A példában használt különösen szegényes programozási stílust azonban nem ajánlott elsajátítani!)

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

/* struktúranév */
struct bla {
   /* struktúra adattag */
   struct bla * bla;
   int cnt;
};

main()
{
     /* közönséges azonosító */
     struct bla * bla;
     int i = 0;

     bla = (struct bla *)malloc(sizeof(bla));
     bla -> cnt = ++i;
     bla -> bla = NULL;
    /* címke */
 bla:
    printf("A \"bla\" blák lánca!\n");
   if (bla -> bla == NULL) {
       bla -> bla =  (struct bla *) malloc(sizeof(bla));
       bla -> bla -> bla = NULL;
       bla -> bla -> cnt = ++i;
       goto bla;
   }
   printf("A ciklus %d iterációja futott le\n",
                                        bla->bla->cnt);
   return (0);
}
```

3.9.5. A tárolási osztályok használata

A tárolási osztály megadásával lehetőségünk van az alapértelmezéstől eltérő élettartam és érvényességi tartomány kialakítására. Azok az azonosítók, amelyek tárolási osztálya **auto** vagy **register**, lokális élettartammal, míg a **static**, illetve **extern** azonosítók globális élettartammal rendelkeznek.

A **typedef** szintén tárolási osztályt jelöl a C nyelvben, azonban csak formai szempontból sorolják ebbe a csoportba. Ezért könyvünkben is külön (az egyes típusok ismertetése után) szerepel.

A változó és a függvény deklarációjának elhelyezkedése a forrás állományban szintén hatással van a tárolási osztályra és a láthatóságra. Azok a deklarációk, amelyek minden függvényen kívül helyezkednek el, a "külső szintű" deklarációk, míg a függvényen belül megadott deklarációk - "belső szintűek".

A tárolási osztály azonosítójának pontos jelentése függ attól, hogy a deklaráció a külső vagy a belső szinten szerepel, illetve attól is, hogy változót vagy függvényt deklarálunk. Az alábbi táblázatban összefoglaltuk az azonosítók élettartamára és láthatóságára vonatkozó megállapításokat:

Jellemzők			**Eredmény**	
Szint	***Tétel***	***Tárolási osztály***	***Élettartam***	***Láthatóság***
file érvényességi tartomány	változó definíció	**static**	globális	korlátozva az adott file-ra, ahol a definíció helyétől a file végéig
	változó deklaráció	**extern**	globális	a definíció helyétől a file végéig
	függvény prototípus vagy definíció	**static**	globális	korlátozva az adott file-ra
	függvény prototípus	**extern**	globális	a definíció helyétől a file végéig
blokk érvényességi tartomány	változó deklaráció	**extern**	globális	blokk
	változó definíció	**static**	globális	blokk
	változó definíció	**auto** vagy **register**	lokális	blokk

Minden tárolási osztály esetén tisztáznunk kell a deklarált változó vagy függvény élettartamát és láthatóságát, illetve változók esetén az inicializálás kérdését is.

3.9.5.1. Az **auto** *tárolási osztály*

Azok a változók, amelyeket blokkon belül definiálunk, alapértelmezés szerint automatikus (**auto**) változók. Az automatikus változók a függvények belső változói, amelyek akkor kezdenek el létezni, amikor a függvényt meghívjuk. A függvényből való kilépés után pedig megszűnnek. (A függvény paramétereit is hasonló módon kezeli a rendszer.)

Nézzünk egy olyan függvényt, amely két változót nagyság szerint sorba állít:

```
void sorbaallit(int * const a, int * const b)
{
   int sv;

   if ( *a > *b ) {
      sv = *a;
      *a = *b;
      *b = sv;
   }
}
```

(A példában a **const** típusminősítő az *a* és *b* mutatók konstans értékét jelöli, így nem használhatók a mutató léptetésére irányuló műveletek.)

Az *sv* automatikus változó, ezért a függvényen kívülről nem érthető el. Az **auto** kulcsszó megadásával közvetlenül előírhatjuk a tárolási osztályt:

```
auto int sv;
```

azonban ezt a formát a gyakorlatban általában nem használjuk.

Az összetett utasítások (a blokkok) egymásba ágyazhatók, és mindegyikben deklarációk és utasítások egyaránt szerepelhetnek:

```
{
  opcionális definíciók és deklarációk

  opcionális utasítássorozat
}
```


Hibátlan programok írása érdekében törekednünk kell arra, hogy a változók definícióját minél közelebb vigyük a felhasználás helyéhez. C nyelven erre egyetlen lehetőség - a blokkok használata - áll a programozó rendelkezésére. Az előző *sorbaallit* függvényt úgy módosítjuk, hogy ez az elv érvényesüljön:

```
void sorbaallit(int * const a, int * const b)
{
   if ( *a > *b ) {
     int sv = *a;
         *a = *b;
         *b = sv;
   }
}
```

A példában látható megoldással az *sv* változó érvényességi tartománya a legbelső blokkra korlátozódott. Minden olyan hivatkozás, amely ezen az összetett utasításon kívül helyezkedik el, hibát eredményez. (Még a függvény hátralevő részéből sem érhető el az *sv*!).

Valamely azonosító ideiglenesen láthatatlanná válik, ha egy belső blokkban ugyanolyan névvel egy másik változót definiálunk. Ez az elfedés a belső blokk végéig terjed, és nincs mód arra, hogy az elfedett (létező) objektumra hivatkozzunk:

```
main()
{
   int sum = 10;

   /* Az int típusú sum értéke 10 */

   {
      float sum=3.1416;

      /* A float típusú sum értéke 3.1416 */

   }

   /* Az int típusú sum értéke 10 */

}
```

Mivel az automatikus változók a blokkból kilépve megszűnnek létezni, ezért alapvetően hibás elgondolás olyan függvényt írni, amely egy automatikus objektum címével tér vissza. Ha a függvényen belül kívánunk helyet foglalni olyan objektum számára, melyet a függvényen kívül használunk, akkor a dinamikus memóriafoglalás eszközeit kell alkalmaznunk.

Az alábbi két függvény közül a *fordit1* hibás (az elmondottak értelmében), míg a *fordit2* a helyes megoldást tartalmazza. A függvények a kapott sztringet megfordítva egy másik területen adják vissza:

```
char * fordit1(const char * s)                    /* !hibás! */
{
   char a[80];     /* Az a tömb automatikus változó! */
   int i = strlen(s);

   a[i]=0;
   while (*s)
         a[--i] = *s++;
   return a;
}

char * fordit2(const char * s)
{
   /* Az a mutató automatikus, de az általa kijelölt */
   /* terület dinamikus helyfoglalású!               */
   char *a = (char *) malloc(strlen(s)+1);
   int i = strlen(s);

   if (!a) return NULL; /* ha nincs hely */
   a[i]=0;
   while (*s)
         a[--i] = *s++;
   return a;
}
```

Az **auto** változók inicializálása minden esetben végbemegy, amikor a vezérlés a blokkhoz kerül. Azonban csak azok az objektumok kapnak kezdőértéket, amelyek definíciójában szerepel kezdőértékadás. (A többi változó értéke határozatlan!). Mivel az automatikus változók esetén az inicializáló kifejezés kiértékelése futási időben történik, ezért tetszőleges kifejezés megadható (például függvényhívás is):

```
int fv(void)
{
   double pi    = asin(1)*2;
   int    lepes = 20;
   double lrad  = 2*pi/lepes;
   int    a;
   int  * pa    = &a;
   . . .
}
```


Meg kell jegyeznünk, hogy az eredeti C verzió nem tette lehetővé az automatikus tömb és struktúra változók inicializálását. Az ANSI C-ben ez megengedett, de csak konstans (fordító által meghatározható) kifejezéseket tartalmazó kezdőértéklista felhasználásával.

3.9.5.2. Az **extern** *tárolási osztály*

A C program általában egy sor külső objektumot használ. A *külső* kifejezést a függvények paramétereit és automatikus változóit jellemző *belső* kifejezéssel ellentétes értelemben használjuk. C nyelv külső azonosítói a függvényeken kívül definiált változók és függvények nevei.

Azok a külső változók és függvények, amelyek definíciójában nem adunk meg tárolási osztályt, alapértelmezés szerint **extern** tárolási osztállyal rendelkeznek. (Természetesen az **extern** közvetlenül is megadható.)

Az **extern** változó és függvények élettartama a programba való belépéstől a program befejezésig terjed. Azonban a láthatósággal lehetnek problémák. Az alábbi példában a ***main*** függvényben is szeretnénk használni azokat a külső definíciókat, amelyet a ***main*** törzse után helyeztünk el a forrás file-ban.

```
main()                                              /* !hibás! */
{
  double pi=3.142567;

  a = 3 *pi;
  kiir(pi);
  kiir(a);
}

int a = 5;                    /* Az a változó definíciója    */

void kiir(int b)             /* A kiir függvény definíciója  */
{
  printf("%d\n", b);
}
```

A program két ok miatt is lefordíthatatlan. Az első hiba az

```
a = 3 *pi;
```

utasításban jelentkezik, hisz a fordító még nem ismeri az *a* változót. (A változókra vonatkozóan semmilyen feltételezéssel nem él a fordító.) A másik

hibát a *kiir* függvény definíciójánál kapjuk, nevezetesen azt, hogy a *kiir* függvény már más típussal deklarált. A C fordító a függvényhívás helyéig nem deklarált függvényeket automatikusan egész visszatérési értékkel rendelkező és tetszőleges számú és típusú argumentummal hívható függvényként deklarálja.

Mindkét hiba egyszerűen korrigálható, ha az *a* és *kiir* külső azonosítókat a program elején deklaráljuk. Alaphelyzetben a változók és a függvény prototípusok érvényességi tartománya a definíció helyétől a file végéig terjed. Azonban a deklaráció megadásával a láthatóságot az egész file-ra kiterjeszthetjük:

```
extern int a;                  /* Az a változó deklarációja */
extern void kiir(int b); /* A kiir függvény prototípusa  */

main()
{
  double pi=3.142567;

  a = 3 *pi;
  kiir(pi);
  kiir(a);
}

int a = 5;                     /* Az a változó definíciója */

void kiir(int b)           /* A kiir függvény definíciója  */
{
  printf("%d\n", b);
}
```

A külső nevek deklarációját külön állományban szokás tárolni, amelyet aztán a program elején beemelünk az **#include** előfordító utasítás segítségével.

Ha a programunk neve EPELDA.C, akkor a külső deklarációkat tartalmazó állományt EPELDA.H néven ajánlott létrehozni. Ezt követően a programunk elején pedig el kell helyezni az alábbi sort:

```
#include "epelda.h"
```

Felhívjuk a figyelmet arra, hogy valamely változó vagy függvény definíciója csak egyetlen egyszer szerepelhet a programban, míg a deklarációinak száma nincs korlátozva.

A külső változók másik alkalmazási lehetősége, hogy alternatívát biztosítsanak a függvények paramétereivel szemben. Vannak esetek (például amikor több függvény ugyanazt az adathalmazt használja), amikor a függvény paraméterek csak elbonyolítják a megoldást. Az ilyen esetekben is külső (globális) változók használata javasolt. A globális változókkal azonban mindig körültekintően kell eljárnunk, mivel potenciális hibaforrást jelentenek. (Ilyen hiba például, amikor a globális *i* változót valamely blokkban újradefiniálás nélkül ciklusváltozónak használhatjuk.)

Az **extern** tárolási osztály azonban a fent bemutatott megoldásoknál jóval több lehetőséget biztosít a programozó számára (a fenti megoldások a **static** tárolási osztállyal is használhatók.)

Már szóltunk róla a 3.2. fejezetben, hogy nagy programok készítése során a forráskódot több file-ban (modulban) elosztva tároljuk. Az egyes modulok között azonban szükség van bizonyos kapcsolatok kialakítására. Az **extern** tárolási osztályú változók és függvények közösek minden modul számára. Ahhoz, hogy elérjünk egy másik file-ban definiált változót vagy függvényt, egyszerűen csak deklarálnunk kell azok azonosítóit.

Az alábbi egyszerű példában két modulból épül fel a programunk. A programban található három függvény mindegyike lépteti a globális *i* változó értékét és kiírja azt a képernyőre. A program főmodulja (a ***main*** függvényt tartalmazó EXTF1.C file):

```
#include <stdio.h>

extern int i;
void kovetkezo (void);
void utolso(void);

main()
{
  i++;
  printf("%d\n",i);     /* i értéke  8 */
  kovetkezo();
}
```

```
int i = 7;              /* i értéke  7 */

void kovetkezo(void)
 {
   i++;
   printf("%d\n",i);  /* i értéke  9 */
   utolso();
}
```

Az *utolso* függvényt tartalmazó modul (EXTF2.C):

```
#include <stdio.h>

extern int i;

void utolso(void)
{
   i++;
   printf("%d\n",i);   /* i értéke 10 */
}
```

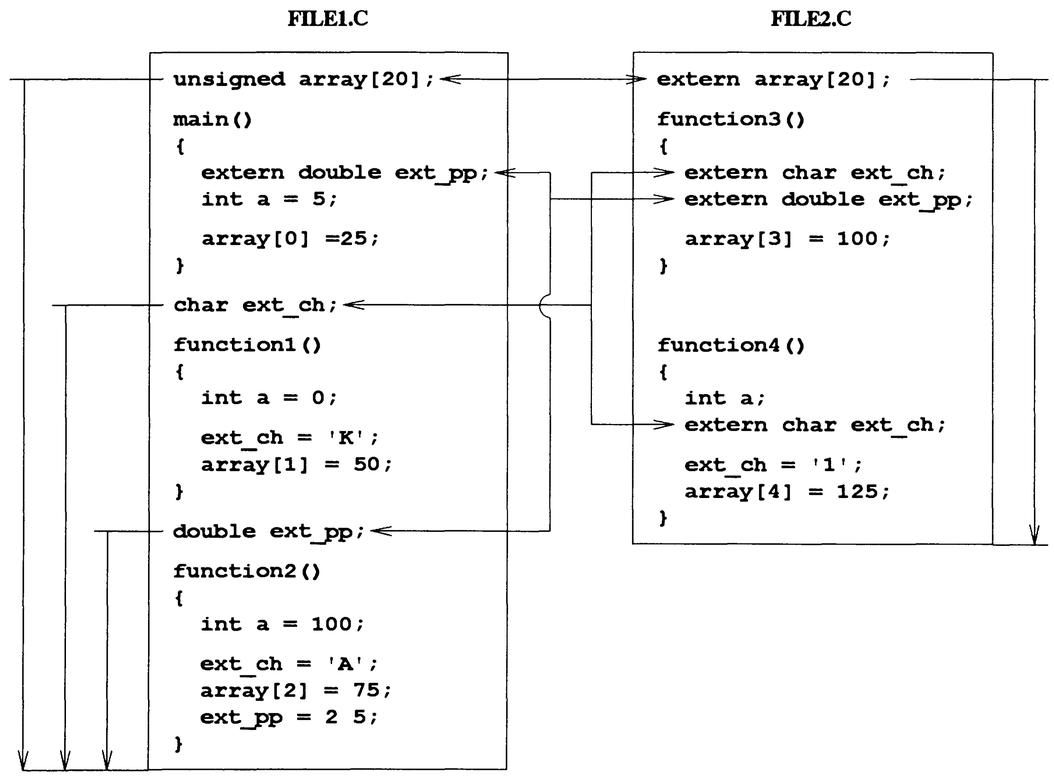
A megfelelő deklarációk elhelyezése után a két modult önállóan le lehet fordítani. Az azonosítók és az objektumok, illetve függvények végső összerendelése a szerkesztő program feladata.

A 3.29. ábrán szintén egy két modulból álló program látható, ahol nyilak segítségével jelöltük az egyes nevek láthatóságát és kapcsolódását.

Az ábrán látható példával kapcsolatban csak annyit szeretnénk megjegyezni, hogy ha az **extern** deklarációt függvényen belülre helyezzük (*function3*), akkor annak láthatósága a függvényre korlátozódik. Ha azonban a file szinten helyezzük el a deklarációt, mint a FILE2.C modul első sora, akkor a név elérhetősége a teljes modulra kiterjed.

Az **extern** változók inicializálása a programba való belépés során egyszer megy végbe. Amennyiben nem adunk meg kezdőértéket a definícióban, úgy a fordító automatikusan 0-val inicializálja az objektumot (feltöltve annak területét nullás byte-okkal). Ha a kezdőértékadásról magunk gondoskodunk, akkor ügyelnünk kell arra, hogy csak a fordító által kiszámítható konstans kifejezést használhatunk:

```
char *p = "Ez az én programom!";
int a[] = { 1,2, 3*4, 4*5};
```

3 29 ábra
Példa modulok összekapcsolására

3.9.5.3. A **static** *tárolási osztály*

A **static** tárolási osztály mind külső, mind pedig belső szintű azonosítókkal együtt használható. Ha külső szintű azonosítók előtt szerepel, akkor az azonosítók láthatóságát a file-ra korlátozza. Ha belső szintű nevek előtt adjuk meg, a nevek élettartamát automatikusról globálisra módosítja.

Amikor több modulból álló programot írunk, a moduláris programozás elvének megvalósításához nem elegendő az, hogy vannak közös változóink. Szükségünk lehet olyan modul szinten definiált változókra és függvényekre is, amelyek elérését a modulra korlátozhatjuk (információ rejtés). Ezért az **extern** és a **static** tárolási osztályú azonosítókat egyaránt használunk a megfelelő modulszerkezet kialakításához.

Példaként készítsünk olyan modult, amely pszeudovéletlen szám előállítására használható. Több modulból álló program esetén feltétlenül szükség van arra, hogy az egyes modulok elején információkat helyezzünk el a file-ok tartalmáról. A modulban definiált **extern** függvények deklarációját a RANDOM.H *include* file tartalmazza.

```
/*
**    File:              random.h
**
**    Pszeudovéletlen számok sorozatának előállítására
**    szolgáló modul globális deklarációi.
*/

extern void set_random(int);
extern int random(void);
```

A RANDOM.C file két kívülről is elérhető függvényt és egy statikus változót tartalmaz:

```
/*
**    File:              random.c
**
**    Pszeudovéletlen számok sorozatának előállítása.
**    set_random - a kiindulási érték beállítása,
**    random     - a következő véletlen szám lekérdezése.
*/
/* konstansok */
#define SZORZO  97
#define OSZTO  256
#define NOVELES 59

/* statikus globális változó definiálása 0 kezdőértékkel */
static unsigned int pseudo;

/* A kiindulási érték beállítása */
void set_random(unsigned int init)
{
  pseudo = init;
}

/* A következő véletlen szám előállítása */
unsigned int random(void)
{
  pseudo = (SZORZO * pseudo + NOVELES) % OSZTO;
  return pseudo;
}
```


Végezetül tekintsük a RANDMAIN.C programot, amely kockadobás szimulációjához használja a fenti *random* rutint:

```
/*
**      A hatoldalú kocka dobásának szimulációja.
**      A példában a kockát 5-ször dobjuk
*/

#include <stdio.h>
#include "random.h"

main()
{
  int i;

  set_random(1994);     /* A generátor inicializálása */
  printf("\nÖtször dobunk a kockával: \n");
  for (i=0; i<5; i++)
   printf("%d. dobás \t%d\n", i, random()%6+1);
}
```

A **static** tárolási osztály másik alkalmazási területe a statikus belső szintű változók kialakítása. Az ilyen változók láthatósága a definíciót tartalmazó blokkra korlátozódik, azonban a változó a program indításától a programból való kilépésig él. A statikus változók inicializálása szintén egyetlen egyszer a változó létrehozásakor megy végbe. Ezért ezek a statikus lokális változók a blokkból való kilépés után (a függvényhívások között) is megőrzik értéküket.

Az előző példánk véletlen szám generátorát egyetlen függvény felhasználásával is megvalósíthatjuk. A módosított *random* függvény használatát a lemezmellékleten található RANDOM2.C programban mutatjuk be. A *random* függvényt:

```
unsigned int random(unsigned int init)
{
  /*statikus lokális változó definiálása 0 kezdőértékkel */
  static unsigned int pseudo;

  if (init)  /* Ha az init nem 0 */
     pseudo = init;
  else
     pseudo = (97 * pseudo + 59) % 256;
  return pseudo;
}
```

kétféleképpen kell hívni. Ha 0-tól különböző argumentummal hívjuk, akkor a véletlen szám sorozat kezdőértékét állítjuk be:

```
random(1994); /* Inicializálás */
```

A következő véletlen szám lekérdezéséhez 0 argumentummal kell a *random* függvény aktivizálnunk:

```
szam = random(0)%6+1;
```

Összefoglalva, a **static** tárolási osztály segítségével olyan azonosítókat deklarálhatunk, melyek élettartama a program futásának teljes idejére kiterjed (permanens). A láthatóság külső azonosítók esetén a definíciót tartalmazó file-ra, míg belső azonosítók esetén a definíciót magában foglaló blokkra korlátozódik.

A statikus változók kezdőértékkel való ellátására ugyanaz a szabály vonatkozik, mint az **extern** változókra. A **static** változók inicializálása a programba való belépés során egyszer megy végbe. Amennyiben nem adunk meg kezdőértéket a definícióban, úgy a fordító automatikusan 0-val inicializálja az objektumot (feltöltve annak területét nullás byte-okkal). Ha a kezdőértékadásról magunk gondoskodunk, akkor ügyelnünk kell arra, hogy csak a fordító által kiszámítható konstans kifejezést használjunk.

3.9.5.4. A **register** *tárolási osztály*

A **register** tárolási osztály csak belső szintű azonosítókhoz (automatikus lokális változókhoz és paraméterekhez) használható. A **register** kulcsszó megadása azt jelöli a fordító számára, hogy az adott változót gyors eléréssel szeretnénk kezelni. Ennek érdekében a fordító (amennyiben módjában áll kérésünket teljesíteni) a processzor regiszterében hozza létre az objektumot. Ha nincs szabad felhasználható regiszter, akkor az objektum **auto** változóként lesz definiálva.

A fenti megoldásból két megállapítás is következik. Az első az, hogy nem használható a "címe" (&) operátor **register** tárolási osztályú operandussal. A másik, hogy nincs mód annak lekérdezésére, hogy a tárolást végül is hol valósította meg a fordító (memóriában vagy regiszterben). A regiszterek adattárolási kapacitása processzoronként eltérő lehet, ezért implementációnként

különböznek azok a típusok, amelyeket regiszterben tárolhatunk. A legtöbb C fordító a **char**, az **int** és a mutató típusokat tárolja regiszterben.

Az alábbi *csere* függvény a korábban bemutatott megoldásnál potenciálisan gyorsabb működésű. (A potenciális szót azért használjuk, mivel nem ismert, hogy a **register** előírások közül hányat teljesít a fordító.)

```
void csere(register int *a, register int *b)
{

    register int c = *a;
                 *a = *b;
                 *b =  c;
}
```

A **register** változók élettartama, láthatósága és inicializálásának módja megegyezik az **auto** tárolási osztályú változókéval.

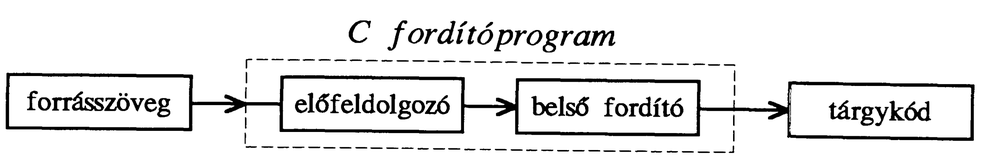
Ellenőrző kérdések

1. Mit takar az azonosítók élettartama?
2. Jellemezze az érvényességi tartomány szintjei!
3. Hol helyezkednek el a programban a külső szintű és a belső szintű deklarációk?
4. Jellemezze az **auto** tárolási osztályt!
5. Melyik tárolási osztály támogatja a moduláris programépítést?
6. Milyen céllal használjuk a **static** tárolási osztályt?
7. Mire szolgál a **register** tárolási osztály?

3.10. Az előfeldolgozó

Minden C fordítóprogram szerves részét képezi az ún. előfeldolgozó *(preprocesszor)*. A fordítóprogram gyakorlatilag nem azt a forráskódot fordítja, amit mi begépelünk, hanem az előfeldolgozó által előállított szöveget dolgozza fel. (A fordítás két fő lépése a 3.30. ábrán követhető nyomon.) A C fordítók többségében ez a két jól elkülöníthető fordítási szakasz nem válik külön, azaz nem jelenik meg az előfordító kimenete valamilyen szöveges állományban. (A Turbo C 2.0 rendszer egy különálló CPP nevű segédprogramot tartalmaz az előfordított program .I kiterjesztésű file-ba töltésére.)

A fentiekben bemutatott működés egyben az előfordító használatának előnyét és hátrányát is jelenti. A fejezetben bemutatásra kerülő megoldások a C program jól olvasható formában történő előállítását támogatják. A hátrány pedig abban áll, hogy a programozó általában nem látja azt az előfordított kódot, amiből a futó program előáll. Ebből következik, hogy bizonyos előfordításból származó programhibák kiderítése nem egyszerű feladat.



3 30 ábra
A C program fordításának lépései

Az előfeldolgozó olyan sororientált szövegfeldolgozó (makronyelv), amely semmit sem "tud" a C nyelvről. Ez két fontos következménnyel 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 csúszhatnak át másik sorba, hacsak nem jelölünk ki folytatósort),
- az előfeldolgozó által elvégzett minden művelet - egyszerű szövegkezelés (függetlenül attól, hogy a C nyelv kulcsszavai, kifejezései vagy változói szerepelnek benne).

A preprocesszornak szóló parancsokat a sor elején (esetleg szóközök és/vagy tabulátorok után) álló # karakter jelzi. Az előfeldolgozót leggyakrabban szöveghelyettesítésre (**#define**) és szöveg file beépítésére (**#include**) használjuk. Ugyancsak jól alkalmazható a program részeinek feltételtől függő fordítására is (**#if**). A preprocesszor parancsokat szokás direktíváknak is nevezni.

3.10.1 A C program fordításának fázisai

Ebben a fejezetben az előfordítóval ismerkedünk meg, azonban érdekességképpen nézzük meg, hogy a két fő fordítási lépés mellett milyen fázisokon keresztül megy végbe a forrásprogram lefordítása. A C program általában több forrásállományt tartalmaz. A forrás file, és az *include* állományok együttesen ún. fordítási egységet (*translation unit*) képeznek.

A futtatható program előállítása során minden fordítási egységet külön le kell fordítani tárgykóddá (*object code*) , amit aztán a szerkesztés során építünk össze egyetlen futtatható állománnyá. Tekintsük sorrendben a fordítás fázisait:

1. A fordító átkódolja a forrásprogramot a C karakterkészletnek megfelelően.
2. Minden olyan sort, amelyet fordított per jel ('\', *backslash*) és újsor karakterek zárnak, egyesít az utána következő sorral.
3. A fordító szétbontja a forrás file-t, preprocesszor tokenekre, tagoló karakterekre és megjegyzésekre. Ezt követően a megjegyzéseket szóköz karakterrel helyettesíti. (Az implementációtól függ, hogy a tagoló karaktereket szóközre cseréli-e vagy sem.)
4. A fordító végrehajt minden előfeldolgozó utasítást és kifejti a makrokat. Minden **#include** direktíva esetén a megadott file beépítésekor végrehajtja az első négy fordítási fázist.
5. A fordító minden *escape*-szekvenciát (a sztring és karakter konstansokban) a futtatási karakterkészlet megfelelő kódjára cseréli.
6. A fordító összekapcsolja az egymás mellett álló sztring literálokat.
7. Ez a fázis az igazi fordítás, amikor minden nyelvi elemet a szintaktikai és szemantikai ellenőrzése után tárgykóddá konvertál a fordító.

8. Az utolsó lépés a külső objektumokra és függvényekre való hivatkozások feloldása és futtatható program előállítása. A fordítási folyamatnak ezt a lépését szerkesztésnek (*linking*) szokták nevezni.

3.10.2. File-ok beépítése a forrásprogramba

Az **#include** direktíva utasítja az előfeldolgozót, hogy az utasításban szereplő szöveges állomány tartalmát építse be a programunkba. (A beépítés helyét a direktíva elhelyezkedése határozza meg.) Általában a deklarációkat és makrokat tartalmazó ún. *header* file-okat (.H) szoktuk beépíteni a programunk elején, azonban tetszőleges szöveg file-ra használható a művelet. A beépítési utasítás általános alakja:

```
#include  <file-név>
```

illetve

```
#include  "file-név"
```

ahol a *file-név* a beépítendő állomány neve. Az első formát általában a C rendszer szabványos *header* file-jainak (mint például az STDIO.H, STDLIB.H, stb.) beépítésére használjuk. A második pedig a saját magunk készített file-ok beépítésére szolgál. (Példaként érdemes visszalapozni a 3.9.5.3 fejezet RANDMAIN.C nevű programjához.)

3.10.3. Makrok használata

A **#define** direktívát arra használjuk, hogy "beszédes" azonosítókkal lássunk el C konstansokat, kulcsszavakat, illetve gyakran használt utasításokat és kifejezéseket. Az így definiált konstansokat reprezentáló makrokat szimbolikus konstansnak (objektumszerű makronak) nevezzük. Általában a kifejezéseket és utasításokat megvalósító definíciókat hívjuk makronak (függvényszerű makro). A makronevekre ugyanaz a képzési szabály vonatkozik, mint más azonosítókra. Azért, hogy a preprocesszor számára definiált szimbólumok a C forrásnyelvi szövegben jól elkülönüljenek a programban használt azonosítóktól, a makroneveket csupa nagybetűvel ajánlott írni.

Az előfeldolgozó minden programsort átvizsgál, hogy az tartalmaz-e valamilyen korábban definiált makronevet. Ha igen, akkor azt lecseréli a megfelelő helyettesítő szövegre, majd újból átvizsgálja a sort további makrokat keresve, amit új helyettesítés követhet. Mindaddig folytatódik ez a folyamat, amíg vagy nem talál a preprocesszor újabb makronevet a sorban, vagy csak olyat talál, amit már egyszer helyettesített (a végtelen rekurziók elkerülése). Ezt a folyamatot makro helyettesítésnek, vagy makro kifejtésnek nevezzük.

A makrokat a **#define** utasítással hozzuk létre, melynek két formája használható. Ha a makrora többé nincs szükségünk, akkor az **#undef** direktíva segítségével megsemmisítjük azt.

3.10.3.1. Objektumszerű makrok készítése

Ebben az esetben a **#define** direktíva egyszerűbb alakját használjuk:

```
#define  azonosító  helyettesítő szöveg
```

Nézzünk néhány példát szimbolikus konstans definiálására (MAKRO1.C):

```
#define EOS         '\0'      /* a sztringvége karakter */
#define TRUE        1
#define FALSE       0
#define NOT         !
#define BOOL        int
#define IGEN        1
#define NEM         !IGEN
#define URES                  /* példa üres makrora */
#define STRING0     ""        /* üres sztring       */
#define UDV         "Üdvözöllek dicsõ lovag ....!"
#define PMERET      1024
#define HA          if
#define KIIR        printf
```

A fenti makrokat használva eléggé furcsa kinézetű (működő) C programot írhatunk. A kifejtés menetének érzékeltetés érdekében nézzük meg a

```
HA (NEM)
```

sor feldolgozásának lépéseit:

```
if (NEM)       if (!IGEN)          if (!1)
```

A program szövegének tanulmányozása után bemutatjuk a preprocesszor kimenetének a ***main*** függvényt tartalmazó részletét.

```
main()
{
    BOOL *p=(BOOL *)malloc(sizeof(BOOL)*PMERET);
    KIIR(UDV);
    HA (NOT IGEN)
        /* Sztringben nem végzi el a helyettesítést! */
        KIIR("URES");
    HA (NEM)
        KIIR(STRING0);
    free(p);
}
```

A makrok behelyettesítése után a C program:

```
main()
{
int  *p=(int  *)malloc(sizeof(int )*1024 );
printf ("Üdvözöllek dicsõ lovag ....!" );
if  (!  1 )
printf ("URES");
if  (!1  )
printf ("" );
free(p);
}
```

3.10.3.2. Függvényszerű makrok készítése

A makrok felhasználási lehetőségeit lényegesen megnövelik a paraméterezett makrok, melyek definíciójának általános formája:

#define *azonosító(paraméterlista) helyettesítő szöveg*

A makro hívása:

azonosító(argumentumlista)

A makrohívásban szereplő argumentumok számának meg kell egyeznie a definícióban szereplő paraméterek számával. (Lehet paraméterek nélküli makrot is definiálni.)

A MAKRO2.C példaprogramban néhány gyakran használt makro definícióját és meghívását mutatjuk be:

```
#include <stdio.h>

/* x abszolút értékének meghatározása */
#define abs(x)    ( (x) < 0 ? (-(x)) : (x) )

/* a és b maximumának kiszámítása */
#define max(a,b) ( (a) > (b) ? (a) : (b) )

/* A és B minimumának kiszámítása */
#define min(A,B) ( (A) < (B) ? (A) : (B) )

/* Négyzetre emelés */
#define sqr(x)    ( (x) * (x) )

/* Két egész objektum tartalmának felcserélése */
/* (csak egész balérték argumentummal működik helyesen) */
#define swap(a,b) { int c; c = a, a=b, b=c; }

main()
{
    int a=-4, b=6;
                                        /* Eredmények */
     printf("%d\t%d\n",a ,b);           /*   -4    6  */
     printf("%d\n", abs(a));            /*    4       */
     printf("%d\n", abs(b));            /*    6       */
     printf("%d\n", sqr(a));            /*   16       */
     printf("%d\n", min(a,b));          /*   -4       */
     printf("%d\n", max(a,b));          /*    6       */
     swap(a,b);
     printf("%d\t%d\n",a ,b);           /*    6   -4  */
}
```

A makrokra jellemző, hogy általában tetszőleges típusú argumentummal meghívhatók. (A fenti *swap* makro kivételnek számít.) A makro törzse az argumentumok aktuális szövegértékének behelyettesítése után bemásolódik a hivatkozás helyére. Nézzük meg közelebről az *sqr(a)* hivatkozást, melyet az előfordító az alábbi kifejezéssel helyettesít:

```
( (a) * (a) )
```

A makro törzsében a paramétereket általában zárójelben kell használnunk, ellenkező esetben bizonyos argumentumokkal aktivizálva hibás eredményt

kapunk. Példaként erre a megállapításra definiáljuk az *sqr* makrot úgy is, hogy ne legyenek zárójelben a paraméterek:

```
#define sqr2(x)    ( x * x )                    /* !hibás! */
```

Hasonlítsuk össze az *sqr* és az *sqr2* hívását *a* és *a+1* argumentumokkal!

	a	***a+1***
sqr	`( (a) * (a) )`	`( (a+1) * (a+1) )`
sqr2	`(a * a)`	~~`(a+1 * a+1)`~~

A zárójelezésnek azonban kellemetlen következményei is lehetnek, mint például amikor valamilyen léptető operátor szerepel a makro argumentumában:

```
a=5;
printf("%d\n", sqr(a++));          /*    30     */
printf("%d\n", a);                 /*     7     */
```

A programrészlet lefutásakor a kiírt értékek *30* és *7*. A hiba a kifejtett makro kiértékelésében keresendő, amely mellékhatást tartalmaz:

```
(a++) * (a++)
```

A mellékhatás miatt a kifejezés kiértékelése implementációfüggő. Sajnos általánosan is elmondható, hogy nem szabad léptető operátort tartalmazó kifejezést makronak átadni. A fenti programrészlet kétféle módon javítható. Az egyszerűbb első megoldást akkor kapjuk, ha a léptetés műveletét külön hajtjuk végre:

```
a=5;
printf("%d\n", sqr(a));            /*    25     */
a++;
printf("%d\n", a);                 /*     6     */
```

A másik szintén jó megoldás, ha a makro helyett függvényt használunk a négyzetre emelés elvégzésére:

```
int square(int x)
{  return x * x;  }
```

Az eddigi példáinkban a helyettesítést csak különálló paraméterek esetén végzi el az előfeldolgozó. Vannak esetek, amikor a paraméter valamely

azonosítónak része, vagy az argumentum értékét sztringként kívánjuk felhasználni. Ha a helyettesítést ekkor is el kívánjuk végezni, akkor a ##, illetve a # makro operátorokat kell használnunk.

A **##** operátor megadásával két szintaktikai egységet (tokent) lehet összeforrasztani oly módon, hogy a makro törzsében a ## operátort helyezzük a paraméterek közé.

A MAKRO3.C példában szereplő *show* makro tetszőleges *x*-szel kezdődő nevű numerikus változó értékét **double** típusúvá konvertálva jeleníti meg:

```
#include <stdio.h>

#define show(a) printf("%lf\n", (double)(x##a))

main()
{
  double x1  = 10;
  int    xyz =20;

  show(yz);     /* printf("%lf\n", (double)(xyz)) ; */
  show(1);      /*  printf("%lf\n", (double)(x1)) ; */
}
```

A # karaktert a makro paramétere elé helyezve, a paraméter értéke idézőjelek között (sztringként) helyettesítődik be. Ezzel a megoldással sztringben is lehetséges a behelyettesítés, hisz a fordító az egymás mellett álló sztring literálokat egyetlen sztring konstanssá kapcsolja össze.

A MAKRO4. C példában az *str* makrot a sztringet megfelelő módon kiíró ***printf*** hívás előállítására használjuk. A *display* makro segítségével tetszőleges változó neve és értéke megjeleníthető. A makro *n* paramétere a változó nevét, míg az *f* a kiírás formátumát jelöli.

```
#include <stdio.h>

#define str(x) printf("\a\n" #x "\n");

#define display(n,f) \
                printf("A változó: " #n "=%" #f "\n",n)
```

```
main()
{
   int     a = 13;
   char   *p = "Hello C";
   double pi = 3.1425;

   str(Most kezdodik);
   display(a,5d);
   display(p,s);
   display(pi,9.6lf);
   str(Vege);
}
```

Összehasonlítás céljából közöljük az előfordító kimentén megjelenő C programot is:

```
main()
{
int     a = 13;
char   *p = "Hello C";
double pi = 3.1425;

printf("\a\n" "Most kezdodik" "\n"); ;
printf("A változó: " "a" "=%" "5d" "\n",a) ;
printf("A változó: " "p" "=%" "s" "\n",p) ;
printf("A változó: " "pi" "=%" "9.6lf" "\n",pi) ;
printf("\a\n" "Vege" "\n"); ;
}
```

Ha a makrodefiníció nem fér el egyetlen sorban, akkor a fordított perjel (*backslash*) felhasználásával folytatósort jelölhetünk ki. A MAKRO5.C példában egy megadott sztringet a képernyősor közepére kiíró több soros makrot definiáltunk (*KOZEPRE*). A kiírás tartományát a BAL_MARGO és a JOBB_MARGO szimbolikus konstansokkal (margók) kell megadni

```
#include <stdio.h>
#include <string.h>

#define BAL_MARGO   10
#define JOBB_MARGO  70
#define HOSSZ       (JOBB_MARGO-BAL_MARGO)
#define KOZEPRE(s)  printf("%*c%s\n", \
                           BAL_MARGO + \
                          ((HOSSZ-strlen(s)) / 2),' ',s);
```



```
main()
{
    char *s = "Középen van! 10 - 70";

   KOZEPRE(s)
   KOZEPRE("Ez is!")
}
```

Ha makro által lefoglalt azonosítót fel szeretnénk szabadítani, akkor az **#undef** direktívát kell használnunk. A makro új tartalommal történő átdefiniálása előtt mindig meg kell szüntetnünk a régi definíciót. Az **#undef** használata:

#undef *azonosító*

Az **#undef** utasítás nem jelez hibát, ha az azonosító nincs definiálva.

3.10.3.3. Előredefiniált makrok

Az ANSI szabványban az alábbi öt előredefiniált makro szerepel:

Makro	Leírás	Példa
__DATE__	A fordítás dátumát tartalmazó sztring konstans.	"Aug 20 1994"
__TIME__	A fordítás időpontját tartalmazó sztring konstans.	"07:30:13"
__FILE__	A forrás file nevét tartalmazó sztring konstans.	"preproc.c"
__LINE__	A forrás file aktuális sorának sorszámát tartalmazó számkonstans (1-től sorszámoz).	123
__STDC__	A makro értéke 1, ha a fordító ANSI C fordítóként működik, különben nem definiált.	

(Mindegyik azonosító két-két aláhúzáskarakterrel kezdődik és végződik.) Az előredefiniált makrok nevét nem lehet sem a **#define** sem pedig az **#undef** utasításokban szerepeltetni.

A legtöbb C implementáció az ANSI által előírt makrokat saját definíciókkal egészíti ki. Példaként nézzük a Turbo C 2.0 rendszer előredefiniált szimbólumait.

Makro	Leírás
__TURBOC__	A Turbo C program verziószáma (2.01 - 0x0201).
__PASCAL__	A makro értéke 1, ha alapértelmezés szerint minden szimbólum a Pascal nyelv konvencióit követi, különben nem definiált.
__MSDOS__	A makro értéke mindig 1.
__CDECL__	A makro értéke 1, ha alapértelmezés szerint minden szimbólum a C nyelv konvencióit követi, különben nem definiált.
__TINY__	A makro értéke 1, ha az alkalmazott memóriamodell a *tiny* modell, különben nem definiált.
__SMALL__	A makro értéke 1, ha az alkalmazott memóriamodell a *small* modell, különben nem definiált.
__MEDIUM__	A makro értéke 1, ha az alkalmazott memóriamodell a *medium* modell, különben nem definiált
__COMPACT__	A makro értéke 1, ha az alkalmazott memóriamodell a *compact* modell, különben nem definiált
__LARGE__	A makro értéke 1, ha az alkalmazott memóriamodell a *large* modell, különben nem definiált.
__HUGE__	A makro értéke 1, ha az alkalmazott memóriamodell a *huge* modell, különben nem definiált.

Az előredefiniált makrok értéke beépíthető a program szövegébe, de a feltételes fordítás feltételeként is felhasználható.

3.10.4. Feltételes fordítás

A feltételes fordítás lehetőségeinek használatával elérhető, hogy a forrásprogram bizonyos részei csak adott feltétel teljesülése esetén kerüljenek be az előfeldolgozó által előállított programba. A feltételesen fordítandó programrészek kijelölésére többféle szerkezet közül választhatunk. A különböző megoldásokat az **#if**, **#ifdef**, **#ifndef**, **#elif**, **#else** és **#endif** preprocesszor direktívákkal állíthatjuk elő. Az **#if** utasításban a feltétel megadására konstans kifejezéseket használhatunk, melyek 0 és nem nulla értéke jelöli az igaz, illetve a hamis feltételt, például:


```
#if 'z' - 'a' == 25
```

A feltételek másik fajtájával azt ellenőrizhetjük, hogy a megadott szimbólum definiált-e vagy sem. Ehhez a feltételben a **defined** operátort használjuk, mely 1 értékkel tér vissza, ha az operandusa létező szimbólum:

```
#if defined szimbolum
```

vagy

```
#if defined(szimbolum)
```

Mivel gyakran használunk ehhez hasonló vizsgálatokat, ezért valamely szimbólum létezésének tesztelésre külön előfeldolgozó utasítás áll a rendelkezésünkre:

```
#ifdef szimbolum
```

Az **#if** segítségével azonban bonyolultabb feltételek is megfogalmazhatók:

```
#if defined szimbolum && ('z' - 'a' == 25)
```

Nézzük először az egyszerűbb szerkezetet, amellyel két programrész közül választhatunk:

```
#if konstans kifejezés
      programrész1
#else
     programrész2
#endif
```

Ez a szerkezet jól használható ún. hibakeresési (*debug*) információk beépítésére a programba. Ekkor egyetlen szimbolikus konstans 1 vagy 0 értékével jelölhetjük ki a lefordítandó programrészeket.

Az alábbi példában szereplő osztást végző függvény a program fejlesztési szakaszában (**#define** DEBUG 1) kiírja a paraméterek értékét, és 0-val való osztás esetén hagyja hibajelzéssel kilépni a programot. Az 'éles' verzióban (**#define** DEBUG 0) azonban valahogy kivédi a 0-val való osztást. A *DEBUG* szimbólum megfelelő értékkel való definiálását a program elején végezzük el:

```
#define DEBUG 1
```

```
double osztas(int a, int b)
{
  #if DEBUG
      printf("osztas - a =%d\n", a);
      printf("osztas - b =%d\n", b);
  #else
      if (b == 0) {
          a = MAXINT;
          b = 1;
      }
  #endif
  return (double)a/b;
}
```

A fenti szerkezet helyett jobb a *DEBUG* szimbólum definiáltságát vizsgáló megoldást használni, melyhez a *DEBUG* érték nélkül is definiálható:

```
#define DEBUG
. . .
#if defined(DEBUG)
    printf("osztas - a =%d\n", a);
    printf("osztas - b =%d\n", b);
#else
    if (b == 0) {
        a = MAXINT;
        b = 1;
    }
#endif
```

Az alábbi két-két vizsgálat ugyanazt az eredményt szolgáltatja:

```
#if defined(DEBUG)
   . . .
#endif
```

```
#ifdef DEBUG
   . . .
#endif
```

illetve a fordított megoldás:

```
#if !defined(DEBUG)
   . . .
#endif
```

```
#ifndef DEBUG
   . . .
#endif
```

Bonyolultabb struktúrák kialakításához egy másik preprocesszor utasítást ajánlott használni, ami többirányú elágaztatás megvalósítását teszi lehetővé. Az utasítás általános formája:


```
#if konstans_kifejezés1
      programrész1
#elif konstans_kifejezés2
      programrész2
#elif konstans_kifejezés3
      programrész3
#else
     programrész4
#endif
```

Az alábbi példában a processzor típusától függően más-más értéket vesz fel az *INTSIZE* szimbolikus konstans:

```
#define I386

#if defined(I8086)
    #define INTSIZE 16
#elif defined(I386)
    #define INTSIZE 32
#elif defined(VAX)
    #define INTSIZE 32
#else
    #define INTSIZE 32
#endif
```

Az **#include** utasítások használata során előfordul, hogy ugyanazt az állományt többször építjük be a programunkba és ez programhibát okoz (Például, ha az STDIO H állományt mind a programunkba, mind pedig a saját H file-ba beépítjük, akkor a saját *header* file beépítése során gondok lehetnek az STDIO H állományban szereplő **#define** utasításokkal.)

Nézzük meg, hogyan védik ki a rendszer *header* állományai a fent vázolt problémát. A következő megoldást minden makrokat definiáló *include* file-ban ajánlott alkalmazni. A H file első beépítésekor létrejön egy szimbólum (pl. *__STDIO_DEF*), melynek létezését vizsgálva elkerülhető az ismételt beépítés:

```
#if !defined(__STDIO_DEF)
#define __STDIO_DEF)

   makro definíciók és egyéb deklarációk

#endif
```

3.10.5. A #line, az #error és a #pragma direktívák

Több olyan segédprogram is létezik, amely valamilyen speciális nyelven megírt programot C forrás programmá alakít át (program generátor). A **#line** direktíva segítségével elérhető, hogy a C fordítóprogram ne a C forrásszövegben jelezze a hiba sorszámát, hanem az eredeti speciális nyelven megírt forrás file-ban. (A **#line** utasítással beállított sorszám és file-név a ***__LINE__*** és a ***__FILE__*** szimbólumok értékében is megjelennek.) A direktíva használata:

```
#line kezdősorszám
```

vagy

```
#line kezdősorszám "file-név"
```

Az **#error** direktívát a programba elhelyezve, fordítási hibaüzenet jeleníthetünk meg, amely az utasításban megadott szöveget tartalmazza. Az utasítás általános formája:

```
#error hibaüzenet
```

Az alábbi példában a fordítás hibaüzenettel zárul, ha a Turbo C rendszerben nem a *small* memóriamodellt állítjuk be:

```
#if !defined(__SMALL__)
    #error A forditas csak small modellel vegezheto!
#endif
```

A **#pragma** direktíva a fordítási folyamat implementációfüggő vezérlésére szolgál. (A direktívához semmilyen szabványos megoldás sem tartozik.) Ha a fordító valamilyen ismeretlen **#pragma** utasítást talál a programban, akkor azt figyelmen kívül hagyja. Ennek következtében a programok hordozhatóságát nem veszélyezteti ez a direktíva. Az utasítás általános alakja:

```
#pragma parancs
```

Példaként tekintsük át a Turbo C 2.0 rendszerben megadható **#pragma** utasításokat!

#pragma direktíva	*Leírás*
`#pragma inline`	Figyelmezteti a fordítót, hogy a programban beépített assembly (**asm**) utasítások szerepelnek
`#pragma warn +nnn`	Engedélyezi az *nnn* sorszámú figyelmeztető üzenet megjelenítését.
`#pragma warn -nnn`	Letiltja az *nnn* sorszámú figyelmeztető üzenet megjelenítését.
`#pragma warn .nnn`	Az *nnn* sorszámú figyelmeztető üzenet megjelenítését alapértelmezés szerint állítja be
`#pragma saveregs`	A direktíva után álló **huge** függvénybe való belépéskor a regiszterek elmentését, a kilépéskor pedig a visszaállítását írja elő.

Ellenőrző kérdések

1. Ismertesse a C forrásprogram fordításának lépéseit!
2. Milyen előfeldolgozó direktíva felhasználásával lehet file-t beépíteni a C programba?
3. Csoportosítsa a makrokat!
4. Mire kell ügyelnünk a függvényszerű makrok írásánál?
5. Milyen előfeldolgozó utasításokkal valósul meg a feltételes fordítás?
6. Mi a szerepe a **#pragma** direktívának?

4. Programozás Turbo C könyvtári függvények felhasználásával

A C nyelv lehetőségeinek többsége a C könyvtári függvényeken keresztül érhető el. A Turbo C rendszer könyvtári függvényeinek teljes csoportosítását az F2. függelék tartalmazza. Ebben a fejezetben a szabványos lehetőségek bemutatásán túlmenően a Borland grafikus rendszerrel foglalkozunk részletesen.

Az egyes témakörökhöz tartozó függvények prototípusát, illetve makrok definícióját külön *header* file-ok tartalmazzák. Ezek közül a szükségeseket a programunk legelején az `#include` előfeldolgozó utasítással kell beépítenünk a programba.

4.1. Alapvető be- és kiviteli függvények

A C nyelv nem rendelkezik adat be- és kiviteli (input/output, I/O) utasításokkal, minden ilyen feladatot könyvtári függvényekkel kell megoldanunk. A függvényeket három csoportra oszthatjuk, melyek közül az első két csoport az operációs rendszer lehetőségeinek felhasználásával végzi el az I/O műveleteket, míg a harmadik csoport közvetlenül a hardver (konzol, BIOS) programozásával.

Az operációs rendszer szintjén a C program számára az adatátvitel mindig adatállomány (file) olvasását és írását jelenti, jóllehet a program bemenete valójában a billentyűzet, kimenete pedig a képernyő vagy a nyomtató. Az adatátvitel egysége a byte, amit a C nyelvben a **char** típus valósít meg.

Az operációs rendszer szintjén választhatunk a szabványos megoldás (adatfolyam - *stream*), illetve az alacsonyszintű (*low-level*) megoldások között. Az MS-DOS lehetőségeinek teljes kihasználásához általában a két szintet együttesen használjuk. Ha azonban hordozható C-forráskódot kívánunk előállítani, akkor a szabványos hívásokra kell hagyatkoznunk.

Az alacsonyszintű I/O függvények lehetővé teszik a file-ok és a perifériális eszközök operációs rendszer szintjén történő elérését. A file-ok azonosítása ezen a szinten a leíróval (*file descriptor, file handle*) történik

Az adatfolyam (*stream*) függvények az állományokat mint karakterek folyamát tekintik. Amikor egy file a *stream*-függvények használatával kerül megnyitásra, a nyitott állományt egy ***FILE*** struktúrával, illetve a ***FILE*** struktúrára mutató pointerrel azonosítja a rendszer. Az adatfolyamot kezelő függvények a formázott vagy formázatlan adatok átvitelét pufferen keresztül vagy puffer használata nélkül képesek lebonyolítani Minden adatfolyamhoz tartozik file-leíró, amit a ***fileno*** függvénnyel lekérdezve, az alacsony szintű függvényeket is felhasználhatjuk a *stream* kezelésére

MS-DOS alatt a C program indításakor öt előredefiniált *stream* automatikusan megnyitódik, lehetővé téve az alapvető perifériális egységek elérését. Az ***stdin*** (szabványos input), ***stdout*** (szabványos output) és ***stderr*** (szabványos hiba) adatfolyam azonosítókat az MS-DOS az ***stdaux*** (szabványos másodlagos I/O) és ***stdprn*** (szabványos nyomtató) nevekkel bővíti

A szabványos input és output adatfolyamok kezeléséhez külön függvények és makrok állnak a programozó rendelkezésére, amelyekkel az alapvető adat be- és kiviteli műveletek elvégezhetők Ezzel a témával a jelen fejezet hátralevő részeiben foglalkozunk

Az adatkezelés másik nagy területe az állományok feldolgozása, melynek szabványos lépéseit a 4.2. fejezetben ismertetjük Az alábbi táblázatban összefoglaltuk a szabványos adatfolyamokat, feltüntetve a hozzájuk tartozó file-leírókat és perifériákat:

Adatfolyam	***Leíró***	***Periféria***	***A művelet iránya***
stdin	0	billentyűzet (átirányítható)	*input*
stdout	1	képernyő (átirányítható)	*output*
stderr	2	képernyő	*output*
stdaux	3	1 kommunikációs csatorna (COM1)	*input/output*
stdprn	4	1 nyomtató port (PRN, LPT1)	*output*

A adat be- és kivitelt végző függvények harmadik csoportját a 4.8 fejezetben tárgyaljuk Ezek a függvények az operációs rendszert megkerülve közvetlenül vagy a ROM BIOS felhasználásával végzik el az I/O műveleteket Ugyancsak itt találhatunk függvényeket, melyek segítségével mód nyílik a szöveges képernyő hatékony kezelésére

A következő táblázatban összefoglaltuk az azonos jellegű input, illetve output műveletek megvalósítására szolgáló Turbo C függvényeket:

STDIO H				CONIOH
stdin	***stdout***	***sztring***	***stream I/0***	***konzol***
getchar	-	-	*getc*	*getch, getche*
-	-	-	*ungetc*	*ungetch*
fgetchar	-	-	*fgetc*	-
gets	-	-	*fgets*	*cgets*
-	-	-	*getw*	-
-	*putchar*	-	*putc*	-
-	*fputchar*	-	*fputc*	-
-	*puts*	-	*fputs*	*cputs*
-	-	-	*putw*	-
-	-	-	*fread*	-
-	-	-	*fwrite*	-
scanf	-	*sscanf*	*fscanf*	*cscanf*
vscanf	-	*vsscanf*	*vfscanf*	-
-	*printf*	*sprintf*	*fprintf*	*cprintf*
-	*vprintf*	*vsprintf*	*vfprintf*	-

A táblázat segítségünkre lehet annak eldöntésében, hogy az adott függvény használata esetén melyik *header* állományt kell beépítenünk a programunkba A jelen és a következő (4 2.) fejezetben bemutatásra kerülő függvények prototípusa az STDIO.H file-ban található:

```
#include <stdio.h>
```

4.1.1. A *getchar* és a *putchar* makrok

A folyam jellegű adatkezelésnél az alapvető művelet egy karakter beolvasása és kiírása, mely feladatok elvégzésre makrokat tartalmaz a szabvány:

```
#define getchar()  getc(stdin)
#define putchar(c) putc((c), stdout)
```

A legegyszerűbb adatbeviteli függvényszerű makro a ***getchar***,

```
int getchar(void);
```

amely egy karaktert olvas a szabványos input eszközről (billentyűzetről). Ha az olvasás sikeres volt, akkor a beolvasott karakter kódját kapjuk meg visszatérési értékként. A file-vége esemény bekövetkeztekor ***EOF*** értékkel tér vissza a ***getchar***. (Billentyűzet esetén a *<Ctrl+Z>* billentyűkombináció váltja ki ezt az eseményt.)

A legegyszerűbb adatkiviteli függvényszerű makro a ***putchar***,

```
int putchar(int ch);
```

amely a megadott *ch* karaktert kiírja szabványos output eszközre (képernyőre) Sikeres működés esetén a *ch* karakter kódját kapjuk meg visszatérési értékként.

Az alábbi CHARIO C példaprogram a *<Ctrl+Z>* és *<Enter>* billentyűk lenyomásáig soronként írja vissza a begépelt karaktereket:

```
#include <stdio.h>
main()
{
   int c;

   while ((c = getchar()) != EOF)
         putchar(c);
}
```

Felhívjuk a figyelmet arra, hogy a beolvasott karakter tárolására **int** típusú változót ajánlott használni, mivel az ***EOF*** értéket (-1) is tárolni kell a 256 lehetséges karakterkód mellett.

4.1.2. A *gets* és *puts* függvények

A C nyelv szabványos könyvtára tartalmaz függvényeket karaktersorozatok (sztringek) egyetlen hívással történő beolvasására (***gets***) és kiírására (***puts***).

```
char* gets(char *sptr);
```

A ***gets*** függvény egy sort (*<Enter>* lenyomásáig) olvas a szabványos inputról, majd a karaktereket az argumentumban megadott *sptr* mutató által kijelölt területre másolja A beolvasott sztringben az újsor ('\n') karakter '\0' (EOS) karakterrel helyettesítődik, ezért a függvény nem alkalmas az újsor karakter beolvasására. (Erre a célra a ***getchar*** makrot kell használnunk.) A ***gets*** visszatérési értéke sikeres olvasás esetén a megadott puffer címe, illetve ***NULL***, ha hiba lép fel.

```
char *puts(char *s);
```

A ***puts*** függvény az argumentumban megadott sztringet a szabványos kimenetre (a képernyőre) írja. A sztring megjelenítést automatikusan a '\n' (újsor) karakter kiírása követi.

A STRINGIO.C program bemutatja a sztringre vonatkozó I/O műveletek felhasználását:

```
#include <stdio.h>
main()
{
  char str[80];

  puts("Sztring beolvasása:");
  gets(str);
  puts(str);
}
```

4.1.3. Formázott adat be- és kivitel

A C nyelv szabványos könyvtára két olyan függvényt (***printf*** és ***scanf***) tartalmaz, amelyek segítségével megadott formátum szerint lehet kiírni, illetve beolvasni alaptípusú adatokat és sztringeket.

4.1.3.1. A printf függvény

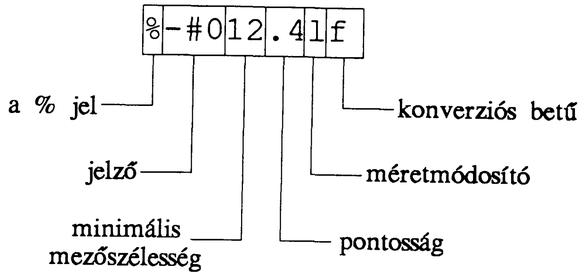
A ***printf*** függvény hívásának formája:

```
printf(formátum, argumentumlista);
```

A *formátum*, amely általában sztring konstans, kétfajta karaktert tartalmazhat. A karakterek első csoportja normál karaktersorozatokat alkotnak a formátum sztringen belül, amelyek minden átalakítás nélkül kerülnek kiírásra.

A karakterek másik csoportja speciális konverziót ír elő, amely egyrészt meghatározza az argumentum értelmezésének módját, másrészt pedig a megjelenítés formáját. A konverziós előírások százalékjellel (%) kezdődnek és valamilyen konverziós karakterrel zárulnak.

A konverziós előírás teljes formájában hat jól elkülöníthető részre osztható:



Az alábbiakban áttekintjük az egyes részek szerepét és a részek megadásának lehetőségeit:

Jelző

- a mínuszjel megadása esetén a konvertált argumentum a mező bal széléhez igazítva jelenik meg,
+ (pluszjel) számok kiírásakor az előjel (+ vagy -) mindig megjelenik,
(szóköz) az előjel helyén a mínuszjel vagy szóköz jelenik meg,
0 a szóköz helyett 0-val töltődik fel a szám előtti szabad terület,
(*hash mark*) az ú.n. alternatív nyomtatási kép jele

Ha a jelző karaktereket elhagyjuk, akkor az output a megadott mezőben jobbra igazítva jelenik meg, és balról szóköz karakterek töltik ki a nem használt területet.

Minimális mezőszélesség

A minimális mezőszélesség decimális jegyeket tartalmazó egész konstans, amely a konvertált argumentum kiírásához határozza meg a mező szélességét Amennyiben az első jegy 0, akkor azt jelzőként értelmezi a rendszer Ha a megadott mezőben nem fér el az eredmény, akkor az a szükséges méretű mezőben fog megjelenni. Ha a csillag (*) karakterrel adjuk meg a mezőszélességet, akkor a tényleges mezőszélesség értéket az argumentumlista soron következő eleme adja meg. Az alábbi két kiírási művelet eredménye ugyanaz lesz:

```
printf("%5d", 125);

int w=5;
printf("*d", w, 125);
```

Pontosság

A formátumban a pont után megadott decimális szám a megjelenítés pontosságát határozza meg. A pontosság értelmezése típusonként eltérő:

- egész számok esetén a jegyek minimális számát (`d`, `i`, `o`, `u`, `x`, `X`),
- lebegőpontos számok esetén tizedesjegyek (`e`, `E`, `f`), illetve az értékes jegyek (`g`, `G`) számát,
- sztring esetén pedig a kiíratni kívánt karakterek maximális számát határozza meg.

Ha a pontosság a specifikációból kimarad, akkor az alapértelmezés szerinti érték (6) lesz figyelembe véve. Ha a pontosságnak 0-át adunk meg, akkor a `d`, `i`, `o`, `u`, `x`, illetve `X` konverziós karakterek esetén az alapértelmezés szerinti számú jegy kerül kiírásra, az `e`, `E` és `f` konverziós karakterek esetén pedig elmarad a tizedespont. A csillag (*) megadása azt jelenti, hogy a tényleges pontossági értéket az argumentumlista soron következő eleme tartalmazza.

Méretmódosító előtag

A méretmódosító betű közvetlenül a konverziós karakter előtt áll. Lehetséges értékei:

- `h` – a `d`, `i`, `o`, `u`, `x` és `X` konverziós karakterrel együtt megadva a **short int** típust jelöli,
- `l` – a `d`, `i`, `o`, `u`, `x` és `X` konverziós karakterrel együtt megadva a **long int** típust jelöli, míg az `e`, `E`, `f`, `g` és `G` betűk esetén a **double** típust (Turbo C),
- `L` – a `e`, `E`, `g`, `G`, `f` konverziós betűkkel együtt használva a **long double** típuselőírásnak felel meg, míg a `d`, `i`, `o`, `u`, `x` és `X` konverziós karakterrel együtt megadva a **long int** típust jelöli,
- `N` – **near** mutató,
- `F` – **far** mutató.

A konverziós operátorok

A formátumelem mindig tartalmaz legalább egy betűt, amely alapján az argumentum feldolgozását és konvertálását végzi a rendszer. (A 4.4. táblázatban összefoglaltuk a lehetséges előírásokat.) Minden konverziós karakternek megfelel valamilyen argumentumtípus és nyomtatási kép.

Nézzünk néhány példát a különböző formátummegadási módokra. A 4.1. táblázatban az *str* sztring kiírásának eredményét foglaltuk össze A kiíráshoz használt formátum-előírást a jobboldali oszlop tartalmazza, míg a nyomtatási kép a baloldali oszlopban tanulmányozható (a | karakter a mezőhatárt jelöli):

```
char str[80];
strcpy(str,"programcsomag");
```

%s	|programcsomag|
%6s	|programcsomag|
%.7s	|program|
%25s	| programcsomag|
%-25s	|programcsomag |
%25.7s	| program|
%-25.7s	|program |

4 1 táblázat
Sztring formázott kiírása

Az egész számok decimális, oktális és hexadecimális formátumban történő kiírását a 4.2. táblázat tartalmazza. A formátumban a kérdőjel (?) helyén a táblázat felső sorában megadott konverziós betűket kell behelyettesíteni (a | karakter a mezőhatárt jelöli):

```
int  j = 4321, m = -4321;
```

	? = d	**? = o**	**? = x**	**? = X**
%?	|4321|	|10341|	|10e1|	|10E1|
%2?	|4321|	|10341|	|10e1|	|10E1|
%6?	| 4321|	|10341d|	| 10e1|	| 10E1|
%-6?	|4321 |	|10341 |	|10e1 |	|10E1 |
%.7?	|0004321|	|0010341|	|00010e1|	|00010E1|
%#?	|4321|	|010341|	|0x10e1|	|0X10E1|

4 2 táblázat
Egész számok formázott kiírása

Lebegőpontos értékek konverziójára 4.3 táblázatban láthatunk példákat (a | karakter a mezőhatárt jelöli):.

```
double  a = 5.12, b = -32.0, c = 0.0000021;
```

	5.12	**-32.0**	**0.0000021**
%lf	|5.120000|	|-32.000000|	|0.000002|
%e	|5.120000e+00|	|-3.200000e+01|	|2.100000e-06|
%g	|5.12|	|-32|	|2.1e-06|
%10.1lf	| 5.1|	| -3.20|	| 0.0|
%+5.0E	|+5E+00|	|-3E+01|	|+2E-06|
%+5.1E	|+5.1E+00|	|-3.2E+01|	|+2.1E-06|
%+7.2E	|+5.12E+00|	|-3.20E+01|	|+2.10E-06|

4 3 táblázat
Lebegőpontos számok formázott kiírása

Konverziós karakter	*Argumentum típusa*	*Nyomtatási kép*
`c`	**int**	**unsigned char** típusra való konvertálás eredményének megfelelő karakter
`d, i`	**int**	Előjeles decimális egész
`u`	**int**	Előjel nélküli decimális egész.
`f`	**double**	*[-]m dddddd*, ahol a *d*-k száma az előírt pontosságtól függ. A pontosság alapértelmezésben 6, a 0 pontossággal a tizedespont nyomtatása elnyomható
`e, E`	**double**	*[-]m dddddd e±xx*, vagy *[-]m ddddddE±xx*, ahol a *d*-k száma az előírt pontosságtól függ
`g, G`	**double**	Ugyanaz, mint a *%e* vagy a *%E*, ha az exponens kisebb, mint -4, vagy nagyobb-egyenlő, mint a *pontosság*, különben pedig a *%f*. Nincsenek vezető 0-k vagy szóközök
`o`	**int**	Előjel nélküli oktális egész
`x, X`	**int**	Hexadecimális egész a 0x vagy 0 előtag nélkül
`p`	**void***	Mutató értéke (implementációfüggő)
`s`	**char***	A sztring karaktereit írja ki az *EOS*-ig, vagy az adott szélesség határáig.
`%`	-	Nincs konverzió - kiírja a % karaktert

4 4 táblázat
*A **printf** függvény formátumának konverziós karakterei*

4.3.1.2. A scanf függvény

Az

```
int scanf(const char *formatum, ...);
```


deklarációjú függvény formázott adatbevitelt valósít meg. A ***scanf*** függvény a szabványos beviteli eszközről olvas be adatokat, és az esetek többségében a ***printf*** konverziók fordítottját hajtja végre. Hívásának általános formája:

```
scanf(formátum, argumentumlista);
```

A ***scanf*** karaktereket olvas a szabványos inputról, majd a formátum sztring specifikációi szerint értelmezi és konvertálja azokat. A konverzió eredményét pedig a soron következő argumentum által kijelölt memóriaterületre tölti.

A ***scanf*** befejezi az olvasást, ha a specifikációnak megfelelő számú adatot már feldolgozott, vagy ha az input valamely oknál fogva nem felel meg a formátum sztring előírásainak. A ***scanf*** visszatérési értéke a sikeresen beolvasott adatok száma, ami jól használható arra, hogy ellenőrizzük, hogy valóban annyi adatot olvasott be a programunk, mint ahányat előírtunk. A 0 visszatérési érték azt jelöli, hogy egyetlen változónk sem kapott értéket a ***scanf*** hívása során.

A formátum sztring felépítése hasonlít a ***printf*** függvénynél használt formátumra, azonban több ponton is különbözik attól. A ***scanf*** formátum sztring az alábbi elemekből épül fel:

- tagoló (*whitespace*) karakterek, melyeket a ***scanf*** figyelmen kívül hagy,
- konverzió specifikációk a % karakterrel kezdődnek és opcionálisan tartalmazhatják a:
  - *** csillag karaktert (melynek hatására a beolvasott adatot a ***scanf*** eldobja),
  - *n* mezőszélességet megadó decimális egész számot,
  - *h, l, L* opcionális méretkijelölő karaktert: `h` (**short**), `l` (**long** vagy **double**), `L` (**long double**) :

  A specifikációt a konverziós karakter zárja, melynek lehetséges értékeit a 4.5. táblázatban foglaltunk össze.

Konverziós karakter	***Input adat***	***Argumentum típusa***
`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***
`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 A 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 tizedespont és az exponens megadása opcionális. Ha a specifikációban az `l` vagy az `L` előtag is szerepel (például *%lf*), akkor az argumentum típusa:	**float*** **double***
`p`	A ***printf*** által kiírt formátumú pointer értékét olvassa be	**void** **
`%`	Nincs hatása.	

4 5 táblázat
*A **scanf** függvény formátumának konverziós karakterei*

A ***scanf*** szűrni is képes a beolvasandó sztring karaktereit. Ha a százalékjel után álló szögletes zárójelben karakterhalmazt adunk meg, akkor csak az ott szereplő karakterek kerülnek be a sztringbe (szűrés).

```
char str[80];

scanf("%[INin]", str);
```

csak a felsorolt karaktereket tartalmazó sztringet olvassa be az input sorból (a példa csak kis- és nagy I, N betűket tartalmazó sztringet olvassa be),

```
scanf("%[0-9a-fA-F]", str);
```

csak a megadott intervallumba eső karaktereket olvassa be (a példa csak hexadecimális jegyeket tartalmazó sztringet fogad el);

```
scanf("%[^abcd]", str);
```

csak azokat a karaktereket olvassa be, amelyek nem szerepelnek a felsorolásban,

```
scanf("%[^0-9]", str);
```

csak a megadott karakter intervallumon kívülre eső karaktereket olvassa be (a példa minden számjegyet nem tartalmazó sztringet elfogad).

Természetesen a karakter halmaz kijelölésénél a fenti eseteket együttesen is használhatjuk.

A ***scanf*** és ***printf*** függvények használatát a FORMIO.C példában mutatjuk be:

```
#include <stdio.h>

main()
{
   double a;
   int    b;
   char szoveg[30];

   printf("Kérem a nevét :");
   scanf("%s", szoveg);
   printf("Kérek két számot vesszõvel elválasztva: ");
   scanf("%lf,%d",&a, &b);
   printf("\nKedves %s!\n",szoveg);
   printf(" A beírt két szám: a = %f \t b = %d\n", a, b);
}
```

4.1.4. Írás sztringbe és olvasás sztringből

C nyelven adott a lehetőség olyan formázott adatátviteli műveletek végzésére, amelyek sztringbe írnak, illetve sztringből olvasnak. A sztringbe való formázott kiíráshoz az ***sprintf*** függvényt használjuk, melynek prototípusa:

```
int sprintf(char* puffer, const char* formatum, ... );
```

A sztringből való olvasás művelete az ***sscanf*** függvénnyel végezhető el:

```
int sscanf(const char* puffer, const char* formatum, ...);
```

A fenti két függvény jól használható különböző típusú számok és sztring közötti konverzióra. A függvények használatát a STRINGIO.C program szemlélteti, ahol a sztring tömbből olvassuk a **double** értékeket, majd kerekített **long** értékként írjuk vissza a sztring tömbbe:

```
#include <stdio.h>

main()
{
   char adat[3][16] = { "12.64","3.7344","2333.23"};
   double e;
   int i;

   for (i=0; i<3; i++) {
       sscanf(adat[i],"%lf",&e);
       e +=0.5;
       sprintf(adat[i], "%ld", (long)e);
   }

   for (i=0; i<3; i++)
       printf("%d. \t %s\n",i ,adat[i]);
}
```

4.1.5. Az *stdio* és *stdout* átirányítása

Az MS-DOS operációs rendszer (a UNIX-hoz hasonlóan) támogatja a szabványos be- és kimenetek átirányítását. A C programunkon belül semmit nem kell tennünk az átirányítás megvalósítása érdekében. Tekintsük újra a CHARIO C példaprogramot, melynek készítsük el a futtatható változatát (CHARIO.EXE)!


```
#include <stdio.h>
main()
{
   int c;

   while ((c = getchar()) != EOF)
         putchar(c);
}
```

Ezek után az MS-DOS készenléti jelnél futtatva a CHARIO.EXE programot, használjuk az MS-DOS átirányító operátorait (<, >, >>):

```
C:\>CHARIO
```

Indítás után a begépelt karaktersort az *<Enter>* lenyomása után visszaírja a képernyőre a program. A kilépéshez a *<Ctrl+Z>* és az *<Enter>* billentyűket kell megnyomnunk.

```
C:\>CHARIO >ADAT.TXT
```

A begépelt karakterek az ADAT.TXT szöveg file-ba másolódnak. A kilépéshez a *<Ctrl+Z>* és az *<Enter>* billentyűket kell megnyomnunk. (Az ***stdout*** átirányítása az ADAT.TXT állományba.)

```
C:\>CHARIO <ADAT.TXT
```

A program megjeleníti a képernyőn az ADAT.TXT szöveg file tartalmát. (Az ***stdin*** átirányítása az ADAT.TXT állományból.)

```
C:\>CHARIO <ADAT.TXT >ADAT2.TXT
```

A program az ADAT.TXT szöveg file tartalmát átmásolja az ADAT2.TXT állományba. (Az ***stdin*** az ADAT.TXT állományt, míg az ***stdout*** ADAT2.TXT állományt jelenti.)

Az átirányítás segítségével interaktív programot is lehet futtatni file-ban tárolt input adatokkal, vagy a program kimenetét file-ba vagy nyomtatóra lehet küldeni.

Feladatok

1. Olvasson be egy max. 5 jegyű egész számot és fordított sorrendben írja vissza a szám jegyeit, az értéktelen nullákat szóközzel helyettesítve (FORDIT.C). Például:

```
  5240
   425
```

2. Olvasson be két egész számot és állapítsa meg, hogy az első szám osztója-e a másodiknak. A program jelezze az oszthatóságot. Ha oszthatóság nem áll fenn, akkor írja ki a hányadost és a maradékot. (OSZTO.C)

3. Olvasson be 4 lebegőpontos számot és számítsa ki az átlagukat, az eredményt *e*, *f* és *g* konverzióval és különböző mezőszélességgel írja ki. (ATLAG.C)

4. Írjon programot amely karaktereket olvas és minden karakter esetén kiírja a karakter bináris és decimális értékét. A program a 'q' karakter leütésére álljon meg. (KARIR.C)

5. Olvasson be egy mondatot, amely ponttal végződik. Írjon programot, amely a beolvasott mondat minden szavát új sorba írja. (A szavak között egy vagy több szóköz áll.) (MONDAT.C)

6. Irjon olyan C programot, amely a szabványos inputról folyamatosan karaktereket olvas az állomány "végéig" és a beolvasott karakterfolyamban előforduló számjegyek közül az utolsó 4 számjegy mértani középértékét kiírja a szabványos kimenetre! (MERTANI.C)

4.2. A szabványos file-kezelés alapjai

A C nyelven az adatállományokat a tartalmuk alapján szöveges és bináris file-okra osztjuk

Szöveges állományok

A szöveges állományok általában olvasható információt tartalmaznak, mint amilyen például a C forrásprogram tárolására használt .C kiterjesztésű file. A szöveges file sorokból épül fel, melyek hossza tetszőleges lehet és minden sor végét a *CR/LF* karakterpár zárja. Az állomány végét szintén speciális vezérlő karakter jelölheti (*EOF* -'\x1A')

A szöveges file-ok feldolgozása során az *EOF* karakter beolvasását a C program file-vége eseményként érzékeli. Mivel a UNIX operációs rendszer alatt a szöveges állományokban a sorokat egyetlen *NL* ('\n') karakter zárja, a C programok az MS-DOS állományok olvasása és írása során konverziót hajtanak végre. Amikor a C program szöveg file-ba ír adatokat, akkor az *NL* karaktert két karakterré (*CR/LF)* alakítja át, illetve olvasáskor a file-ban található *CR/LF* karaktereket egyetlen *NL* karakterre cseréli

A 4.1. fejezetben használt szabványos adatfolyamok azonosítói szöveges állományokat jelölnek

Bináris állományok

A bináris állomány byte-okból felépülő adathalmaz, amelyben minden karaktert a 8 bites kódja alapján értelmezünk Természetesen a szöveges file mindig feldolgozható mint bináris file

Alapvető különbség van a két file-típus között a különböző típusú számok tárolásában. Ha a **short int** típusú 1994 számot szöveges állományba írjuk (pl. ***fprinf***), akkor olvasható formában az '1', '9', '9', '4' karakterek sorozatával kerül tárolásra. Ugyanezt a számot bináris file-ba írva (pl. ***putw***), a szám a memóriában tárolt formában (2 byte-on) kerül kiírásra.

Ebben a fejezetben a file-kezelés lépéseivel ismerkedünk meg, amelyek C nyelven sem különböznek a más nyelveken használt műveletektől:

1 a file-t azonosító mutató definiálása,
2. a file megnyitása,
3 írás a file-ba, olvasás a file-ból és pozicionálás a file-ban,
4 a file lezárása.

4.2.1. A file-mutató deklarálása

A file-ok adatfolyamként történő kezelése során egy ***FILE*** * típusú ún. file-mutató azonosítja a file-t. (A ***FILE*** struktúrát az STDIO.H file deklarálja.) A definíció formája:

```
#include <stdio.h>
.
.
.
FILE * fp;
```

Minden programban felhasználni kívánt adatállományhoz külön file-mutatót kell létrehoznunk.

4.2.2. A file megnyitása

Ahhoz, hogy a háttértáron levő file tartalmához hozzáférjünk, a file-t meg kell nyitnunk A megnyitás egyrészt kapcsolatot teremt a file-mutató és az operációs rendszer adott állománya között, másrészt pedig a feldolgozáshoz definiálja a file típusát (szöveges vagy bináris) és a hozzáférés módját.

A file megnyitását az ***fopen*** függvény hívásával végezhetjük el, melynek prototípusa:

```
FILE * fopen(const char * filename, const char *mode);
```

Híváskor a *filename* argumentum helyén az állomány operációs rendszer által használt nevét tartalmazó sztringet kell megadni. Felhívjuk a figyelmet az teljes elérési útvonal definiálásakor használt *backslah* karakter sztring konstansban való megadására:

```
"C:\\WORK\\LEVEL.TXT"
```


A *mode* argumentum, amely a file elérését és típusát határozza meg, szintén sztring A *mode* sztringben a karakterek első csoportja az elérési módot definiálja. A lehetséges elérési módokat a 4.6. táblázat tartalmazza.

`"r'`	-	Létező file megnyitása olvasásra.
`"w"`	-	Új file megnyitása írásra. Ha a file már létezik, akkor a tartalma elvész
`"a"`	-	File megnyitása hozzáírásra. A nyitás után a file végén lesz az aktuális file-pozíció Ha a file nem létezik, akkor az ***fopen*** létrehozza azt.
`"r+"`	-	Létező file megnyitása írásra és olvasásra (*update*)
`"w+"`	-	Új file megnyitása írásra és olvasásra (*update*). Ha a file már létezik, akkor a tartalma elvész.
`"a+"`	-	File megnyitása a file végén végzett írásra és olvasásra (*update*). Ha a file nem létezik, akkor az ***fopen*** létrehozza azt

4 6 táblázat
*Az **fopen** hívásban megadható megnyitási módok*

A *w* és a *w+* módot körültekintően kell használni, hisz ha a megnyitni kívánt állomány már létezik, akkor annak tartalma, a file 0 hosszúra való rövidítése során, elvész. A + jel az aktualizálás (*update*) műveletet jelöli, mely az írást és az olvasást egyaránt engedélyezi, Felhívjuk azonban a figyelmet arra, hogy a két művelet váltásakor a puffereket a file-ba kell íratni. Ebből a célból az ***fflush***, vagy pedig a pozicionálást végző ***fseek***, ***fsetpos*** vagy ***rewind*** függvények valamelyikét meg kell hívnunk.

A file típusát meghatározó *t* (*text*) és *b* (*binary*) betűk egyike a mód karakterek után következik a *mode* sztringben. Ha nem adjuk meg a file típusát, akkor automatikusan az ***_fmode*** globális rendszerváltozó értékének megfelelő típusú file nyílik meg. (Alapértelmezés szerint az ***_fmode*** értéke ***O_TEXT***, vagyis szöveges file A bináris file-t az ***O_BINARY*** konstans definiálja. A konstansokat az FCNTL.H állomány tartalmazza.)

Az ***fopen*** függvény visszatérési értéke ***FILE*** * típusú mutató, amely sikeres megnyitás esetén kijelöli a file-hoz tartozó ***FILE*** struktúrát a memóriában. Ha a megadott állományt nem sikerült megnyitni, vagy ha a ***FILE*** struktúrának nem sikerült helyet foglalni a memóriában, ***NULL*** lesz a

függvényérték. Az alábbi programrészlet bemutatja a file-nyitás helyes használatát:

```
FILE * fp;

fp = fopen("CBOOK.TXT", "rt");
if (fp == NULL) {
   fprintf(stderr, "Sikertelen file-nyitási kisérlet!\n");
   exit(-1);
}
```

Az ***fopen*** hívását és a vizsgálatot gyakran egybeépítve használjuk:

```
if (!(fp = fopen("CBOOK.TXT", "rt"))) {
   fprintf(stderr, "Sikertelen file-nyitási kisérlet!\n");
   exit(-1);
}
```

A file sikeres megnyitását a file-kezelésének lépései követik Azonban először a file lezárásával ismerkedünk meg

4.2.3. A file lezárása

Amikor többé nincs szükségünk a megnyitott file-ra, akkor kell használnunk az ***fclose*** hívást, amely lezárja a file-t A file lezárás együtt jár a memóriában a file-hoz tartozó pufferek és más területek felszabadításával. Amennyiben a megnyitott állományon írási műveleteket hajtottunk végre, lezárás előtt ajánlott a memóriában található átmeneti pufferek tartalmát a file-ba írni, az ***fflush*** függvény segítségével. Mindkét függvény a file-mutatót várja az argumentumában:

```
fflush(fp);
fclose(fp);
```

4.2.4. Adatátviteli pufferek kijelölése

A memória és a háttértár közötti adatforgalom gyorsítása érdekében minden *stream*-hez automatikusan egy puffer kerül lefoglalásra, melynek méretét a ***BUFSIZ*** makro határozza meg (512 byte). Közvetlenül file megnyitása után adott a lehetőség arra, hogy a beavatkozzunk a puffer használatába

A ***setbuf*** függvény segítségével saját területünkön jelölhetjük ki a ***BUFSIZ*** méretű puffert:

```
setbuf(FILE * fp, char * puffer);
```

A *puffer* argumentumban a ***NULL*** pointert átadva letilthatjuk a puffer használatát:

```
setbuf( stdout, NULL);
```

A ***setvbuf*** függvény még több lehetőséget biztosít számunkra:

```
setvbuf(FILE * fp, char * puffer, int mod, size_t meret);
```

A *meret* méretű (max 32767 byte) puffer számára területet mi magunk is foglalhatunk (***malloc***), de ha a *puffer* argumentum értéke ***NULL***, akkor a rendszer végzi el a foglalást és a felszabadítást (IBM PC gépeken a puffer méretét a szektorméret többszöröseként ajánlott megadni: *N x 512* byte.)

A *mod* argumentum a puffer használatának módját definiálja. Az argumentum lehetséges értékeit szimbolikus konstansok tartalmazzák:

_IOFBF
: Teljesen pufferelt mód. Ekkor a teljes puffer kiolvasása/feltöltése után kerül sor a puffer újbóli feltöltésére/lemezre írására (olvasás/írás).

_IOLBF
: Soronként pufferelt mód. A file olvasásakor csak a puffer kiürülése után töltődik újra a file-ból puffer, azonban írás esetén minden kiírt sor (*NL*) után kikerül a puffer tartalma a file-ba.

_IONBF
: Nem használ puffert a rendszer. Ekkor a *puffer* és a *meret* paraméterek értéke figyelmen kívül marad.

A 0 függvényérték jelzi a művelet sikerességét. Az alábbi példában letiltjuk az ***stdin*** számára a pufferhasználatot:

```
if (setvbuf( stdin, NULL, _IONBF, 0))
   fprintf(stderr, "\nEgyedül nem megy!\n");
```

4.2.5. Szöveges file kezelése

Mint már említettük, a 4.1. fejezetben használt szabványos adatfolyamok szöveges file-t jelölnek a rendszer számára Ezért az ott bemutatott I/O függvények megfelelő változata minden további nélkül használható tetszőleges szöveges file-ként megnyitott adatfolyam esetén. Nézzük sorra a függvények prototípusát:

– *Karakter olvasása és írása*

```
int getc(FILE *stream);

int putc(int c, FILE *stream);
```

– *Sztring olvasása és írása*

```
char *gets(char *s);

int puts(const char *s);
```

– *Formázott adatok olvasása és írása*

```
int fscanf (FILE *stream,  const char *format,...);

int fprintf (FILE *stream, const char *format, ...);
```

Az alábbi TEXTFILE.C példaprogram a formázott műveleteket mutatja be:

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

main()
{
 FILE * tf;
 int i, a;
 double sum, adat;

/* Az ADATOK.TXT file létrehozása és feltöltése sorokkal */
 tf = fopen("ADATOK.TXT", "wt");
 if (tf==NULL) {
    fprintf(stderr, "Sikertelen file-nyitási kisérlet!\n");
    exit(-1);
 }
```



```
 for (i=0; i<10; i++)
     fprintf(tf, "%4d,%10.8f\n", i, i*3.14159265);
 fflush(tf);
 fclose(tf);

/* Az ADATOK.TXT file tartalmának visszaolvasása */
 if (!(tf = fopen("adatok.txt", "rt"))) {
    fprintf(stderr, "Sikertelen file-nyitási kisérlet!\n");
    exit(-1);
 }
 sum = i = 0;
 while (!feof(tf)){
       fscanf(tf,"%d,%lf", &a, &adat);
       sum += adat;
       i += a;
       printf("%d. adatsor feldolgozása megtörtént!\n", a);
 }
 fclose(tf);

 printf("A %d db. PI átlaga: %10.8f\n", i, sum/i);

}
```

A példában a visszaolvasásra használt módszer tetszőleges adatsort tartalmazó file esetén működik. File olvasásakor az ***feof*** függvény 0 értéke jelzi a file-vége esemény bekövetkezését:

```
int feof(FILE * fp);
```

4.2.6. Bináris file kezelése

A bináris állományok kezelését byte-onként, vagy pedig blokkonként végezhetjük el. A byte-os kezelés megvalósítására kényelmes lehetőséget biztosít a ***getc*** és a ***putc*** függvénypáros. A memóriablokkok file-ba írására és file-ból történő visszaolvasására az ***fwrite***, illetve az ***fread*** függvények használhatók, melyek prototípusa:

```
size_t fwrite(const void *ptr, size_t size, size_t n,
              FILE*stream);

size_t fread(void *ptr, size_t size, size_t n,
             FILE *stream);
```

Mindkét függvény adott *size* méretű adatelemekkel dolgozik, melyekből *n* darabot ír vagy olvas a hívás során. Az adatokat tartalmazó memóriaterület kezdetét a *ptr* mutató jelöli ki. Felhívjuk a figyelmet, hogy TURBO C rendszerben a *size*n* kifejezésnek, vagyis a feldolgozandó memóriaterület méretének, kisebbnek kell lenni mint 64 Kbyte.

A függvények visszatérési értéke a kiírt, illetve a beolvasott elemek számát (nem byte-ok!) adja meg. Ha ez az érték kisebb, mint az általunk kijelölt (*n*), akkor file-vége, vagy file-hiba eseményre lehet következtetni.

Mivel C-ben nem létezik a rekordos file fogalma, ezért a rekordok írására és olvasására is a fenti memóriablokkal működő függvényeket használjuk, mint ahogy az a BINFILE.C példában is látható:

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

typedef struct {
        char nev[30];
        int kor;
      } RECORD;

RECORD rt[10], re;

main()
{
    FILE * bf;
    int cnt;

    /* A 0. és a 10  elem feltöltése */
    strcpy(rt[0].nev, "AT&T Bell Lab.");
    rt[0].kor = 25;

    strcpy(rt[10].nev, "K&R - AT&T Bell Lab.");
    rt[10].kor = 35;

    /* A rekordtömb kiírása bináris file-ba */
    bf = fopen("ADAT.DAT", "w+b");
    if (bf==NULL) {
       fprintf(stderr, "Sikertelen file-nyitási"
               " kisérlet!\n");
       exit(-1);
    }
    cnt = fwrite(rt,sizeof(RECORD),100, bf);
```



```
    /* Visszapozicionálás a file elejére */
    rewind(bf);

    /* A 0. rekord visszaolvasása */
    cnt = fread(&re,sizeof(RECORD),1, bf);
    if (cnt!=1) {
       fprintf(stderr, "File olvasási hiba!\n");
       exit(-1);
    }

    /* Pozicionálás a 10. rekordra */
    fseek(bf, 10 * sizeof(RECORD), SEEK_SET);

    /* A 10. rekord visszaolvasása */
    cnt = fread(&re,sizeof(RECORD),1, bf);
    if (cnt!=1) {
       fprintf(stderr, "File olvasási hiba!\n");
       exit(-1);
    }

    fclose(bf);
}
```

4.2.7. Pozicionálás a file-ban

Az adatfolyamok tárolási egysége a byte, ezért a file-okban mindig 0-tól kezdődő byte pozícióra lehet pozicionálni.

A file elejére való pozicionálás a ***rewind*** függvénnyel végezhető el legkényelmesebben. Tetszőleges pozíció kiválasztására az ***fseek***, az aktuális pozíció lekérdezésére pedig az ***ftell*** függvényeket használhatjuk:

```
int fseek(FILE *stream, long offset, int honnan);

long ftell(FILE *stream);
```

Az ***fseek*** hívásánál az *offset* paraméter a file-pozíció relatív távolságát tartalmazza, a *honnan* által kijelölt pozícióhoz képest:

SEEK_CUR	-	az aktuális pozícióhoz képest,
SEEK_END	-	az file végéhez képest,
SEEK_SET	-	az file elejéhez képest.

4.2.8. Hibakezelés

A file-kezelés során többféle hiba léphet fel, a nyitási hibától kezdve, a file-vége esemény érzékeléséig A hibák kezelésére több függvényt is használhatunk:

feof(*fp*)	0 visszatérési értékkel jelzi, hogy elértük a file végét,
ferrof(*fp*)	a *stream* hibajelző adatát teszteli, és 0 értékkel tér vissza, ha hiba volt az utolsó file-művelet során,
clearerr(*fp*)	a hívás alaphelyzetbe állítja a *stream* hiba- és file-vége jelzőjét.

Feladatok

1. Tervezzen olyan C programot, amely egy tetszőleges C forrásállományból kiszűri a megjegyzéseket! (A C nyelvben a megjegyzések /* és a */ karaktersorok által határolt szövegrészek). A megjegyzések kiíratását a szűrőprogram őrizze meg. A program a feldolgozandó forrásállományt a szabványos bemenetről vegye, a megjegyzéstől megfosztott szöveget pedig a szabványos kimenetre írja ki! Feltételezheti, hogy a C forrásállomány nem tartalmaz egymásba ágyazott megjegyzéseket. (KOMMENT.C)

2 Írjon C programot, amely statisztikát készít arról, hogy egy szöveges file-ban az egyes szavak milyen gyakorisággal fordulnak elő. A statisztikát a szavak ABC sorrendje szerinti rendben írja ki tabuláltan a szabványos kimenetre!
Feltételezhető, hogy egy szó nem hosszabb 20 karakternél. Szónak tekinthető minden olyan karaktersor, amelyet tetszőleges számú ú.n. *blank* karakter (szóköz, '\n', vagy tabulátor) határol. A feldolgozandó állományt jellemző file pointert paraméterként vegye át a függvény. A megoldáshoz definiáljon alkalmas adatstruktúrát! A megírandó függvény a hívó programmal csak paraméterlistáján keresztül kommunikálhat. Ügyeljen arra, hogy a függvényből való kilépésekor minden dinamikusan foglalt memóriaterületet szabadítson fel! (STAT.C)

3. A program repülővel utazó utasokat tart nyilván. Az alábbi adatokat kell beolvasni és a REP.TXT file-ba írni:

```
NEV           max. 20 karakteres sztring
UTLEVELSZAM   egész szám
SZ_DATUM      egész szám
R_DATUM       ee/hh/nn (8 karakter)
KM            egész szám
```

Írjon programot, amely az adatokat beolvassa és kiviszi a REP.TXT nevű file-ba! Minden rekordot újsor karakter zár le. A mezőket egymástól egy szóköz karakter választja el. (REPIR.C)

Írjon olyan C programot, amely beolvassa a REP.TXT file-t, az adatokat a memóriában láncolva tárolja és az alábbi feldolgozást végzi (REPOLV.C):

a. / ABC sorrendben kiírja azokat az utazókat és útlevél számukat, akik 10000 km-nél többet repültek,

b. / a legtöbbet repült utazó adatait,

c. / a legfiatalabb utazó nevét és életkorát, valamint születési évét.

4. A program paraméterként olvassa be a file nevet (BE.TXT), másolja át karakterenként nagybetűre átalakítva egy olyan kimenő file-ba, melynek a neve megegyezik a bemenő file nevével és a kiterjesztése pedig BE.OUT legyen! (MASOL.C)

5. Készítsen olyan C programot, amely a BEMENO.TXT nevű szöveges ASCII file első n sorát abc sorrendbe rendezi és a rendezett sorokat a KIMENO.TXT állományba másolja! Az n értékét a klaviatúráról olvassa be. Ha a beolvasott adatnál kevesebb sor van a bemenő file-ban, akkor a program írjon ki egy figyelmeztető üzenetet. (A rendezéshez használja fel a ***qsort*** könyvtári függvényt!) (SORREND.C)

4.3. Karakterek osztályozása - adatkonverzió

4.3.1. Karakterek osztályozása

A Turbo C szabványos könyvtára gazdag készletét tartalmazza az egyetlen karaktert feldolgozó makroknak. A karaktert kezelő makrokhoz és függvényekhez a CTYPE.H *include* file-t kell használni:

```
#include <ctype.h>
```

A C nyelv a karaktereket két csoportba osztja:

- ***Nyomtatható karakterek:***
  ezeket a számítógép meg tudja jeleníteni a képernyőn. A nyomtatható karakterek a szóköz (0x20) és a hullámvonal (~ 0x7E) karakterek között helyezkednek el.
- ***Vezérlő karakterek:***
  ezeknek kódja a (0) és az (0x1F) intervallumba esik, de ide tartozik a DEL (0x7F) karakter is.

A karaktert tesztelő függvényszerű makrok **int** típusú argumentummal rendelkeznek, amelynek a makrok csak az alsó byte-ját használják fel.

```
int isalnum(int c);
```

Az ***isalnum*** makro visszatérési értéke nem nulla, ha a *c* értéke betű ('A'-'Z', 'a'-'z') vagy számjegy ('0'-'9').

```
int isalfa(int c);
```

Az ***isalfa*** makro visszatérési értéke nem nulla, ha a *c* értéke betű ('A'-'Z', 'a'-'z').

```
int isascii(int c);
```

Az ***isascii*** makro visszatérési értéke nem nulla, ha a *c* alsó byte-jának értéke 0 és 127 (0x00-0x7F) közé esik.


```
int iscntrl(int c);
```

Az ***iscntrl*** makro visszatérési értéke nem nulla, ha a *c* értéke vezérlő karakter.

```
int isdigit(int c);
```

Az ***isdigit*** makro visszatérési értéke nem nulla, ha a *c* értéke számjegy ('0'-'9').

```
int isgraph(int c);
```

Az ***isgraph*** makro visszatérési értéke nem nulla, ha a *c* nyomtatható karakter, de nem szóköz.

```
int islower(int c);
```

Az ***islower*** makro visszatérési értéke nem nulla, ha a *c* értéke kisbetű ('a'-'z').

```
int isprint(int c);
```

Az ***isprint*** makro visszatérési értéke nem nulla, ha a *c* nyomtatható karakter.

```
int ispunct(int c);
```

Az ***ispunct*** makro visszatérési értéke nem nulla, ha a *c* elválasztó karakter. Elválasztó karakternek számít az összes nyomtatható karakter a betűk, számok és a szóköz nélkül.

```
int isspace(int c);
```

Az ***isspace*** makro visszatérési értéke nem nulla, ha a *c* szóköz, kocsivissza, újsor, vízszintes és függőleges tabulátor vagy lapdobás karakter (0x09-0x0D, 0x20).

```
int isupper(int c);
```

Az ***isupper*** makro visszatérési értéke nem nulla, ha a *c* nagybetű ('A'-'Z').

```
int isxdigit(int c);
```

Az ***isxdigit*** makro visszatérési értéke nem nulla, ha a *c* hexadecimális számjegy ('0'-'9', 'A'-'F', 'a'-'f').

4.3.2. Karaktert átalakító függvények és makrok

```
int toascii(int c);
```

A ***toascii*** makro ASCII karakterré alakít (az alsó 7 bit kivételével törli a c összes bitjét). Visszatérési érték a *c* konvertált értéke.

```
int tolower(int c);
```

A ***tolower*** függvény az angol nagybetűket kisbetűre alakítja. Visszatérési érték a *c* konvertált értéke.

```
int toupper(int c);
```

A ***toupper*** függvény az angol kisbetűket nagybetűre alakítja. Visszatérési érték a *c* konvertált értéke.

4.3.3. Konverziós függvények

Az alábbi könyvtári függvények a különböző típusú numerikus értékek és sztring között végeznek konverziókat. (Erre a célra természetesen használhatjuk a 4.1. fejezetben ismertetett ***sscanf*** és ***sprintf*** függvényeket is.) A konverziós függvények prototípusát az STDLIB H *include* file tartalmazza:

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

4.3.3.1 Sztring átalakítása numerikus értékké

```
double atof(const char *s);
```

Az ***atof*** függvény sztringet konvertál át lebegőpontos (**double**) értékké Ha a konvertálás sikertelen, a visszatérési érték 0.


```
int atoi(const char *s);
```

Az ***atoi*** függvény sztringet konvertál egész számmá. Ha a konvertálás sikertelen, a visszatérési érték 0.

```
long atol(const char *s);
```

Az ***atol*** sztringet konvertál hosszú egész számmá. Ha a konvertálás sikertelen, a visszatérési értéké 0L.

```
double strtod(const char *s, char **endptr);
```

Az ***strtod*** függvény az *s* sztringet duplapontosságú számmá alakítja át A függvény visszatérési értéke a duplapontosságú szám. Túlcsordulás esetén a visszatérési érték a ***HUGE_VAL*** Ha az átalakítás hibás, akkor a **endptr* tartalmazza a hibás karakterre mutató pointert.

```
long strtol(const char *s, char **endptr, int radix);
```

Az ***strtol*** az *s* sztringet **long** értékké alakítja, és függvényértékként visszaadja A *radix* a konvertálás alapszámát határozza meg (2-36) Ha a *radix* 0, akkor az *s* sztring első néhány karaktere határozza meg a konvertálás alapját:

Első karakter	*Második karakter*	*A sztring értelmezése*
'0'	'1' - '7'	oktális
'0'	'x' vagy 'X'	hexadecimális
'1' - '9'		decimális

Ha az átalakítás hibás, a **endptr* tartalmazza a hibás karakterre mutató pointert

```
unsigned long strtoul(const char *s, char **endptr,
                      int radix);
```

Az ***strtoul*** függvény sztringet **unsigned long** értékre alakít A visszatérési érték **unsigned long** típusú szám. Ha az átalakítás hibás, a **endptr* tartalmazza a hibás karakterre mutató pointert.

4.3.3.2 Numerikus értékek sztringgé átalakítása

```
char *itoa(int value, char *string, int radix);
```

Az ***itoa*** függvény egész számot sztringgé konvertál. A *value* értékét EOS karakterrel lezárt sztringgé konvertálja a *string* mutatta tömbbe. A *radix* a konvertálás számrendszerét határozza meg (2-36). Ha a *value* negatív, és *radix* 10, akkor előjelesen konvertál, egyébként előjel nélkül. A visszatérési érték a sztringre mutató pointer.

```
char *ltoa(int value, char *string, int radix);
```

Az ***ltoa*** függvény egész számot sztringgé konvertál. A *value* értékét EOS karakterrel lezárt sztringgé konvertálja a *string* mutatta tömbbe. A *radix* a konvertálás számrendszerét határozza meg (2-36). Ha a *value* negatív és *radix* 10, akkor előjelesen konvertál, egyébként előjel nélkül. A visszatérési érték a sztringre mutató pointer.

```
char *ultoa(unsigned long value, char *string,
                        int radix);
```

Az ***ultoa*** függvény az **unsigned long** *value* értékét sztringgé konvertálja. A *radix* a konvertálás számrendszerét határozza meg (2-36).

```
char *ecvt(double value, int ndig, int *dec, int *sign);
```

Az ***ecvt*** függvény a *value* lebegőpontos számot *ndig* számjegyet tartalmazó sztringgé konvertálja a tizedespont és az előjel elhelyezése nélkül A **dec* változóba a tizedespont pozícióját teszi a rutin, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a sztringre mutató pointer, ami egy statikus pufferre mutat, amit az ***ecvt*** következő hívása felülír.

```
char *fcvt(double value, int ndig, int *dec, int *sign);
```

Az ***fcvt*** függvény a *value* lebegőpontos számot *ndig* számjegyet tartalmazó sztringgé konvertálja a tizedespont és az előjel elhelyezése nélkül (FORTRAN formátumban). A **dec* változóba a tizedespont pozícióját teszi a rutin, a **sign* változóba pedig nem 0 kerül, ha az érték negatív. A visszatérési érték a sztringre mutató pointer, ami egy statikus pufferre mutat, amit az ***fcvt*** következő hívása felülír.


```
char *gcvt(double value, int ndec, char *buf);
```

A ***gcvt*** függvény a lebegőpontos *value* számot sztringgé konvertálja tizedestört alakban. A *value* értékét konvertálja *ndec* értékes számjegyre, fixpontos alakban, ha lehet, egyébként lebegőpontos alakban. Az eredmény a *buf* mutatta pufferbe kerül, EOS karakterrel lezárva.

4.4. Puffer- és sztringkezelő függvények

4.4.1. Pufferek kezelése

A puffer kifejezés alatt a memóriának adott címén kezdődő, adott hosszúságú területét értjük. IBM PC számítógépen a pufferek maximális hossza 64K. A pufferek kezeléséhez szükséges függvények prototípusát a MEM.H állomány (STRING.H is) tartalmazza:

```
#include <mem.h>

void *memchr(const void *s, int c, size_t n);
```

A ***memchr*** függvény egy mutatót ad vissza, amely a *c*-ben adott karakter első előfordulási helyét jelzi az *s* pufferben. (***NULL*** jelzi, ha a karakter nincs benne.)

```
void *memcmp(const void *s1, const void *s2, size_t n);
```

A ***memcmp*** függvény az *s1* és *s2* puffer első *n* byte-ját hasonlítja össze. A visszatérési érték negatív, ha *s1<s2*, nulla, ha *s1==s2* és pozitív, ha *s1>s2*.

```
void *memcpy(void *dest, const void *src, size_t n);
```

A ***memcpy*** függvény az *src* pufferből *n* byte-nyi blokkot másol a *dest* területre. Átfedésben levő területek esetén nem alkalmazható.

```
int memicmp(const void *s1, const void *s2, size_t n);
```

A ***memicmp*** függvény az *s1 és s2* puffer *n* byte-ját hasonlítja össze, de a kis- és a nagybetűk között nem tesz különbséget.

```
void *memmove(void *dest, const void *src, size_t n);
```

A ***memmove*** függvény az *src* pufferből *n* byte-ot másol a *dest* területre. Egymást átfedő pufferek esetén is jól működik.

```
void *memset(void *s, int c, size_t n);
```

A ***memset*** függvény az *s* puffer *n* byte-ját feltölti a *c* karakterrel.

4.4.2. Sztringek kezelése

A C-ben a sztring '\0' karekterrel záródó karaktertömböt jelent. A sztringkezelő függvények prototípusát a STRING.H file tartalmazza:

```
#include <string.h>

char *strcat(char *str1, const char *str2);
```

Az ***strcat*** függvény az *str1* sztringhez fűzi az *str2* sztringet és az összefűzött sztringre mutató pointerrel tér vissza

```
char *strchr(const char *str1, int c);
```

Az ***strchr*** függvény megkeresi a *c*-ben tárolt karakter első előfordulási helyét az *str1* sztringben és erre a pozícióra mutató pointerre tér vissza. (A ***NULL*** függvényérték jelzi, hogy a karakter nincs benne a sztringben.)

```
char *strcmp(const char *str1, const char *str2);
```

Az ***strcmp*** függvény két sztringet hasonlít össze. A visszatérési érték negatív, ha *str1<str2*, nulla, ha *str1==str2* és pozitív, ha *str1>str2*

```
char *strcmpi(const char *str1, const char *str2);
```

Az ***strcmpi*** függvény két sztringet hasonlít össze, a kis- és nagybetűk megkülönböztetése nélkül. A visszatérési érték negatív, ha *str1<str2*, nulla, ha *str1==str2* és pozitív, ha *str1>str2*

```
char *strcpy(char *str1, const char *str2);
```

Az ***strcpy*** függvény az *str2* sztring karaktereit másolja az *str1* sztringbe a lezáró EOS ('\0') karakterrel együtt.

```
char *strdup(const char *str);
```

Az ***strdup*** az *str* sztringet másolja egy dinamikusan foglalt tárterületre. A visszatérési érték a másolatra mutató pointer, illetve ***NULL***, ha nem sikerült a helyfoglalás.

```
size_t strlen(const char *str);
```

Az ***strlen*** függvény a sztring hosszát adja vissza. A visszatérési érték az *str* sztring hossza a lezáró EOS karakter nélkül.

```
char *strlwr(char *str);
```

Az ***strlwr*** függvény az *str* sztring angol nagybetűit kisbetűkre cseréli le.

```
char *strncat(const char *str1,
              const char *str2, size_t maxn);
```

Az ***strncat*** függvény az *str2* sztringből *maxn* karaktert fűz az *str1* sztring végéhez.

```
char *strncmp(const char *str1,
              const char *str2, size_t maxn);
```

Az ***strncmp*** függvény az összehasonlításnál mindkét sztringből csak *maxn* darab karaktert vesz figyelembe. A visszatérési érték negatív, ha *str1<str2*, nulla, ha *str1==str2* és pozitív, ha *str1>str2*

```
char *strpbkr(const char *str1, const char str2);
```

Az ***strpbkr*** függvény egy sztringben egy másik sztring bármely karakterének első előfordulási helyét keresi, a visszatérési értéke az *str2* bármely karakterének az *str1*-beli első előfordulási helyére mutató pointer. Ha az *str2* semelyik karaktere nem található az *str1*-ben, ***NULL*** értékkel tér vissza.

```
char *strrchr(char *str, int c);
```

Az ***strrchr*** függvény egy sztringben adott karakter utolsó előfordulási helyét keresi. A visszatérési értéke a *c*-ben tárolt karakter *str*-beli utolsó előfordulási helyére mutató pointer Ha a karakter nem található az *str*-ben, akkor ***NULL*** értékkel tér vissza.

```
char *strrev(char *str);
```

Az ***strrev*** függvény helyben megfordít egy sztringet, de az EOS karaktert a helyén hagyja. A visszatérési érték az *str* megfordított sztringre mutató pointer.


```
char *strset(char *str, int c);
```

Az ***strset*** függvény a *c*-ben tárolt karakterrel feltölti az *str* sztringet. A visszatérési érték az *str*-re mutató pointer.

```
char *strstr(const char *str1, const char *str2);
```

Az ***strstr*** függvény az *str2* részlánc első előfordulási helyét keresi az *str1* sztringben. A visszatérési érték az *str2* részlánc *str1*-beli első előfordulási helyére mutató pointer. Ha az *str2* nem található az *str1*-ben, ***NULL*** értékkel tér vissza.

```
char *strtok(char *str1, const char *str2);
```

A ***strtok*** függvény az *str2* sztringben megadott elválasztójel alapján többszöri hívással feldarabolja az *str1* sztringet. Minden visszatérési érték egy-egy részsztringre mutató pointer (vagy ***NULL***, ha az eredeti sztring nem darabolható tovább).

4.5 Matematikai függvények

A Turbo C könyvtára különféle matematikai függvényeket tartalmaz, amelyek többsége **double** típusú argumentummal és visszatérési értékkel rendelkezik. A függvények hibás működésüket az *errno* változón keresztül jelzik:

EDOM - *Domain error*
RANGE - *Result out of range*

Ebben a fejezetben a következő függvénycsoportokat ismertetjük:

- trigonometikus függvények,
- hiperbolikus függvények,
- exponenciális és logaritmikus függvények,
- különféle egyéb függvények.

A matematikai függvények használata esetén a MATH.H *include* file-t szükséges a programunkba beépíteni:

```
#include <math.h>
```

4.5.1. Trigonometrikus függvények

```
double acos(double arg)
```

Az ***acos*** függvény az *arg* arkusz koszinusz értékével tér vissza. Az argumentumnak -1 és 1 között kell lennie, máskülönben "*domain error*" hiba lép fel A függvényértékek a 0 és π között helyezkednek el.

```
double asin(double arg)
```

Az ***asin*** függvény az *arg* arkusz szinusz értékével tér vissza Az argumentumnak -1 és 1 között kell lennie, máskülönben "*domain error*" hiba lép fel. A függvényértékek -π/2 és π/2 között helyezkednek el
Az ***acos*** és ***asin*** függvények hibás argumentum megadás esetén 0 értékkel térnek vissza és az *errno* hibaváltozó értéke ***EDOM*** lesz.


```
double atan(double arg)
```

Az ***atan*** függvény az *arg* arkusz tangens értékével tér vissza. A függvényértékek a π/2 és π/2 közé esnek.

```
double atan2(double y, double x)
```

Az ***atan2*** függvény az *y*/*x* arkusz tangens értékével tér vissza. A függvény felhasználja az argumentum előjelét a visszatérési érték térnegyedének meghatározásához. A függvény -π és π között szolgáltat értékeket és akkor is helyes eredményt ad, amikor az eredmény szög π/2, illetve -π/2 közelében található (*x* 0-hoz közel van). Ha az *x* és az *y* értéke 0, akkor az *errno* változó értéke ***EDOM*** lesz.

```
double cos(double arg)
```

A ***cos*** függvény az *arg* koszinusz értékével tér vissza. Az argumentum értékét radiánban kell megadni.

```
double sin(double arg)
```

A ***sin*** függvény az *arg* szinusz értékével tér vissza. Az argumentum értékét radiánban kell megadni.

```
double tan(double arg)
```

A ***tan*** függvény az *arg* tangens értékével tér vissza. Az argumentum értékét radiánban kell megadni. Ha az argumentum π/2 vagy -π/2 értékekhez közel van, a ***tan*** függvény 0 értékkel tér vissza, és az *errno* változó értéke ***ERANGE*** lesz.

4.5.2. Hiperbolikus függvények

```
double cosh(double arg)
```

A ***cosh*** függvény az *arg* koszinusz hiperbolikusz értékével tér vissza.

```
double sinh(double arg)
```

A ***sinh*** függvény az *arg* szinusz hiperbolikusz értékével tér vissza.

```
double tanh(double arg)
```

A ***tanh*** függvény az *arg* tangens hiperbolikusz értékével tér vissza. A ***cosh*** és ***sinh*** függvények túlcsordulás esetén *HUGE_VAL* értékkel térnek vissza (előjelhelyesen), és az *errno* felveszi az ***ERANGE*** értéket.

4.5.3. Exponenciális és logaritmikus függvények

```
double exp(double arg)
```

Az ***exp*** függvény az e^{arg} kifejezés értékével tér vissza. Túlcsordulás esetén a függvény értéke a *HUGE_VAL*, míg alulcsordulás esetén 0.0 lesz. Mindkét esetet az *errno* változó ***ERANGE*** értéke jelzi.

```
double log(double num)
```

A ***log*** függvény a *num* argumentum természetes alapú logaritmusával tér vissza. "*Domain error*" üzenetet kapunk, ha a *num* argumentum előjele negatív, és "*Range error*" hibaüzenetet kapunk, ha az argumentum zérus.

```
double log10(double num)
```

A ***log10*** függvény a *num* argumentum tízes alapú logaritmusával tér vissza. Ha a *num* értéke 0 vagy negatív, akkor mindkét logaritmus függvény esetén az *errno* változó értéke ***EDOM*** lesz.

4.5.4. Különféle egyéb függvények

4.5.4.1. Abszolútérték függvények

```
int abs(int num)
```

Az ***abs*** függvény a **int** *num* abszolút értékével tér vissza.

```
double cabs(struct complex z)
```


A ***cabs*** függvény a *z* komplex szám abszolút értékével tér vissza. A struktúra a MATH.H *include* file-ban van definiálva.

```
double fabs(double num)
```

Az ***fabs*** függvény a **double** *num* abszolút értékével tér vissza.

```
long labs(long num)
```

Az ***labs*** függvény a **long** *num* abszolút értékével tér vissza.

4.5.4.2. Hányados és maradék függvények

```
#include <stdlib.h>
div_t div(int x, int y)
```

A ***div*** függvény két egészet oszt el, és a ***div_t*** típusú struktúrában visszaadja meg a hányadost (*quot*) és a maradékot (*rem*).

```
double ceil(double num)
```

A ***ceil*** függvény a legkisebb olyan egész számmal tér vissza, amelyik nem kisebb a *num* értékénél Például *ceil(2 15)* 3.0 és *ceil(-2 15)* pedig -2.0 értéket ad vissza

```
double floor(double num)
```

A ***floor*** függvény a legnagyobb olyan egész számmal tér vissza, amely nem nagyobb *num* értékénél. Például *floor(2 15)* esetén 2.0 és *floor(-2 15)* esetén -3.0 lesz a visszatérési érték.

```
double modf(double x, int *ipart)
```

A ***modf*** függvény az *x*-et szétvágja két részre: egy egészre, amelyet az *ipart*-ban tárol, és a törtrészre, amit függvényértékként ad vissza.

```
double fmod(double x, double y)
```

Az ***fmod*** függvény az *x modulo y*-t (az x/y maradékot) adja vissza. A függvény az $x = iy+f$ kifejezést állítja elő, ahol az *i* egész szám, míg az *f* a függvény által visszaadott maradék.

```
double pow(double x, double y)
```

A ***pow*** függvény az *x* argumentum *y* hatványkitevőre emelt értékével (x^y) tér vissza. Túlcsordulás esetén a függvény értéke ***HUGE_VAL***, és az ***errno*** változó értéke ***ERANGE***. Ha az $x \leq 0$ és *y* nem egész, akkor ***EDOM*** hiba keletkezik és a függvény értéke ***HUGE_VAL*** lesz. Ha *x* és *y* értéke 0 (0^0) a függvény értéke 1.

```
double sqrt(double num)
```

Az ***sqrt*** függvény a *num* négyzetgyökének értékével tér vissza. Ha az argumentum negatív előjelű, akkor az *errno* változó értéke ***EDOM***, és 0.0 értékkel tér vissza az ***sqrt***.

```
#include <float.h>
void _fpreset(void)
```

Az ***_fpreset*** függvény a lebegőpontos rutincsomagot újrainicializálja.

```
#include <stdlib.h>
int rand(void)
```

A ***rand*** függvény álvéletlen számot ad vissza ***0*** és ***RAND_MAX*** ($2^{15} - 1$) között.

```
#include <stdlib.h>
int random(int num)
```

A ***random*** függvény álvéletlen számot ad vissza 0 és *num*-1 között.

```
#include <time.h>
#include <stdlib.h>
void randomize(void)
```

A ***randimoze*** makro a pszeudovéletlen sorozatot inicializálja.


```
#include <stdlib.h>
void srand(unsigned seed)
```

Az ***srand*** függvény pszeudovéletlen sorozatot inicializál. Argumetum nélküli hívásnál 1-re lesz beállítva a kezdőpont, egyébként a *seed* segítségével adhatjuk meg a kívánt értékre

4.6. Memóriakezelés Turbo C rendszerben

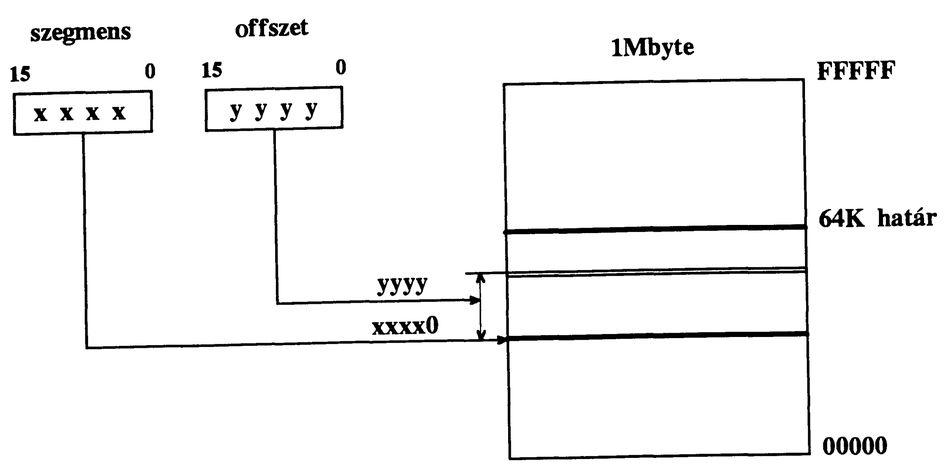
4.6.1. Memóriamodellek

Ahhoz, hogy megértsük a memóriamodellek jelentőségét, mélyen a C nyelv színfalai mögé kell néznünk: meg kell ismerkednünk az IBM PC számítógép lelkével a 8086 jelű mikroprocesszorral.

A 8086 processzor 20 bites fizikai címeket használva címezi a maximálisan 1 MByte méretű memóriát. Ezt a 20 bites címet két 16 bites értékből - a szegmenscímből és az offszetből - állítja elő a mikroprocesszor (4.7. ábra):

```
fizikai_cím = (szegmenscím << 4 ) + offszet;
```

A szegmenscím olyan tartományát jelöli ki a memóriának, amely 16-tal osztható fizikai címen (paragrafus határon) kezdődik és maximálisan 64 KByte mérettel rendelkezik. Ezen terület kezdetéhez képest határozza meg az offszet a kérdéses byte, szó, stb. elhelyezkedését.



4 7 ábra
A 8086 processzor memóriacímzése

A processzor egyidejűleg négy szegmenst tud elérni a négy szegmensregiszter felhasználásával. A ***CS*** regiszter által kijelölt kódszegmens a futó program kódját tartalmazza. A program adatait tartalmazó szegmenseket a ***DS*** és az

ES regiszterekkel érhetjük el. Speciális tulajdonságú adatszegmens a *stack* (verem), amelyet az ***SS*** regiszter definiál A fenti regiszterek tartalma megváltoztatható, így lehetőség van arra, hogy a futó program egy vagy több kód- vagy adatszegmenst használjon.

Milyen hatással van ez a memóriacímzés a C program kialakítására az MS-DOS operációs rendszerben?

Nap mint nap beleütközünk a 64 KByte-os határba. Nem csak a fordító által létrehozott objektumoknak kell kisebbnek lenni mint 64K, de egyetlen könyvtári függvény sem képes 64 KByte-nál nagyobb területet kezelni.

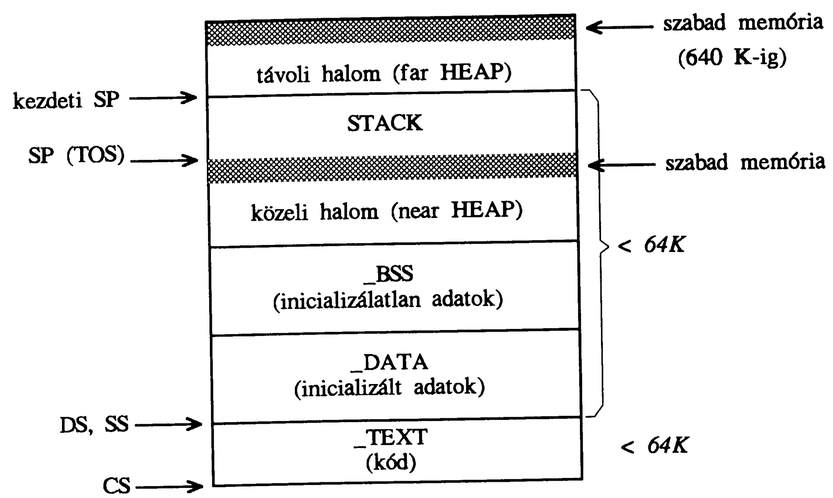
A futó C program területét három fő részre oszthatjuk: kód, adat (*stack* is) és dinamikus terület. A kód és az adatterület viszonya programonként változik, hiszen lehet írni kis kódú programot, amely sok adattal dolgozik, de olyat is, amelynek kódja elég nagy méretű, de minimális adatot használ. Az MS-DOS alatt működő C fordítóprogramok a memóriamodellek bevezetésével lehetővé teszik, hogy a programunk számára a megfelelő kód-adat viszonyt állítsuk be A kód, illetve adatok összterülete lehet 64KByte-nál nagyobb is, amennyiben az több szegmensben helyezkedik el. A négy lehetséges esetet táblázatba rendezve, megkapjuk a négy alap memóriamodellt:

	Adat < 64K (egy szegmens)	**Adat > 64K (több szegmens)**
Kód < 64K (egy szegmens)	*small* modell	*compact* modell
Kód > 64K (több szegmens)	*medium* modell	*large* modell

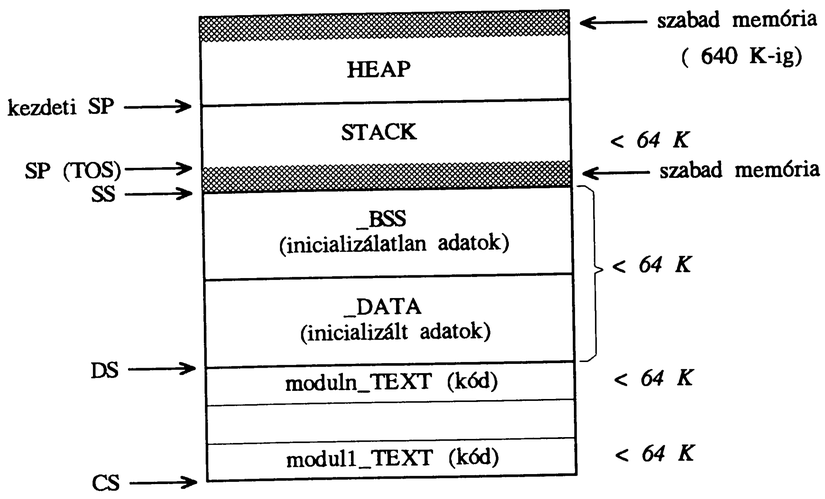
A Turbo C a fenti négy memóriamodell mellett a *tiny* és a *huge* modellek használatát is támogatja A *tiny* modellt választva a program kódja és adatai egyetlen szegmensben tárolódnak. (Az ilyen módon előállított .EXE futtatható állomány .COM file-lá konvertálható.) A *large* modell esetén a program statikus adatainak összmérete továbbra is egy szegmensre (az adatszegmensre) korlátozódik, míg a *huge* modellt használva az egyes modulok statikus adatai

külön szegmensekbe kerülnek. A *large* és a *medium* modelleknél minden modul kódja külön szegmensbe kerül.

A 4.8. ábrán a *small* modell, míg a 4.9. ábrán a *large* modell memória-kiosztása látható.



4 8 ábra
A small modell memória-kiosztása



4 9 ábra
A large modell memória-kiosztása

A C program a különböző élettartamú objektumokat különböző területeken tárolja. Az adatszegmensben a kezdőértékkel ellátott (_DATA) és a kezdőérték nélküli (_BSS) statikus élettartamú adatok helyezkednek el. Az automatikus élettartamú adatok a veremben (STACK-en) jönnek létre. A halomterület (HEAP) pedig a dinamikus élettartamú adatok tárolására szolgál. *Small* és *medium* modellek esetén mind a közeli (adatszegmensben lévő), mind pedig a távoli *heap* elérhető.

A memóriamodell a fenti memória-kiosztás mellett a mutatók méretét is meghatározza. Amennyiben egyetlen kód- vagy adatszegmenst tartalmaz a programunk, a függvényekre, illetve az objektumokra való hivatkozás megoldható a 16 bites offszettel (közeli - **near**). Ha azonban több szegmenst használ a programunk, akkor a szegmenscímet is tartalmaznia kell a hivatkozásnak, ami így 32 bites érték lesz (távoli - **far**) Az alábbi táblázatban összefoglaltuk az egyes memóriamodellekhez tartozó mutató típusokat:

Memóriamodell	***Függvény mutató***	***Adat mutató***
tiny	**near, _cs**	**near, _ds**
small	**near, _cs**	**near, _ds**
medium	**far**	**near, _ds**
compact	**near, _cs**	**far**
large	**far**	**far**
huge	**far**	**far**

A Turbo C rendszer által bevezetett új **near**, **far**, **huge**, **_cs**, **_ds**, **_es** és **_ss** kulcsszavak mint a mutatók típusmódosítói használhatók:

```
int near * pa;
int far * fp = 0xb8000000L;
char huge * hp = fp;
```

A **near** típusú mutatóval csak az adatszegmensen belül mozoghatunk A **far** mutatóval a memória tetszőleges szegmensét megcímezhetjük és abban mozoghatunk. A **huge** mutatóval pedig a teljes 1 Mbyte-os címtartomány

folytonosan lépkedve bejárható. (A **huge** mutató szintén 32 bites címet tárol, azonban ún. normalizált formában.)

A **_cs**, **_ds**, **_es** és **_ss** típusmódosítók szintén **near** mutatókat határoznak meg, amelyek a módosító nevében szereplő szegmensben lesznek értelmezve. A

```
char _ss *p;
```

definíció után a *p* mutatóval a program *stack*-jében érhetünk el adatokat.

A **near** és a **far** típusmódosítókat függvények deklarációjában is megadhatjuk. Az alábbi példában tetszőleges memóriamodell esetén a rekurzív *inpower* függvény a kisebb helyigényű és gyorsabb **near** függvényként fog lefutni. A *medium*, *large* és *huge* modellek esetén az *inpower* függvényt fizikailag nem lehet más modulból meghívni, ezért ajánlott azt statikusnak deklarálni. Ahhoz, hogy mégis aktivizálni lehessen az *inpower* függvényt, egy kapcsolódási függvényt alakítunk ki *power* néven (NPOWER.C):

```
#include <stdio.h>

/*  A hatvány függvény prototípusa */
static double near inpower(double, int);

/* Interface az inpower függvényhez: a memóriamodelltől
   függ a függvény címének a típusa! */
double power(double, int);

main()
{
   int i;

   for (i=-5; i<6; i++)
       printf("2 ^%3d = %8.4lf\n", i, power(2.0,i));
}

/* Az interface függvény definíciója */
double power(double x, int exp)
{
  return inpower(x, exp);
}
```



```
static double near inpower(double x, int exp)
{

   if (!exp || !x)      /* ha 0 a kitevõ vagy 0.0 az alap */
      return 1;
   else
   {
     if (exp < 0) {     /* ha negatív a kitevõ */
         x = 1 /x;
         exp = -exp;
      }
      return x*power(x, exp-1);
   }
}
```

4.6.2. A dinamikus memóriakezelés függvényei

A dinamikus memóriakezelés elvégzéséhez szükséges függvények prototípusát a szabványos STDLIB.H *include* file tartalmazza. (Turbo C rendszerben a prototípusok az ALLOC.H állományban is megtalalhatók.)

Tekintsük először a szabványos megoldásokat. A szükséges memóriablokk lefoglalását kétféleképpen is elvégezhetjük a ***malloc*** és a ***calloc*** függvények segítségével:

```
void *malloc(size_t m);
```

A ***malloc*** függvény *m* byte méretű memóriablokkot lefoglal a halomterületen. A függvény visszatérési értéke az újonnan lefoglalt memóriablokkra mutató pointer.

```
void *calloc(size_t n, size_t meret);
```

A ***calloc*** függvény lefoglal egy *n*meret* byte méretű memóriaterületet, majd pedig feltölti 0 értékű byte-okkal (kinullázza). Visszatérési értéke szintén a memóriablokkra mutató pointer.

```
void *realloc(void *ptr, size_t m);
```

A ***realloc*** függvény újrafoglalással módosítja a már lefoglalt memóriablokk méretét. A *ptr* argumentum által kijelölt memóriablokk méretét *m* byte-ra módosítja a függvény. Ha a blokkot növelni kell, akkor szükség esetén a régi blokk tartalmát átmásolja az új helyre. A visszatérési érték az újonnan

lefoglalt memóriablokkra mutató pointer. (Ha a *ptr* ***NULL*** pointer, akkor a ***malloc*** függvénynek megfelelően működik a ***realloc*** függvény.)

Mindhárom függvény ***NULL*** (nulla) mutató visszatérési értékkel jelzi, ha nem sikerült elvégeznie a kijelölt műveletet, ezért a programunkban mindig gondoskodnunk kell a függvényérték vizsgálatáról. Turbo C-ben a lefoglalható legnagyobb blokk méretét a ***size_t*** típus (**unsigned int**) szabja meg (64K).

Ha egy memóriablokkra nincs többé szükségünk, akkor azt fel kell szabadítanunk, átengedve annak területét a későbbi foglalási műveleteknek:

```
void free(void *ptr);
```

A ***free*** függvény felszabadítja a *ptr* mutató által kijelölt memóriablokkot, amelyet előzőleg a ***calloc,*** a ***malloc*** vagy a ***realloc*** hívások valamelyikével foglaltunk le.

A Turbo C egy információs függvénnyel (***coreleft***) bővíti a szabványos függvényeket, amely megadja a halomterület végén található szabad memóriablokk méretét. Mivel a foglalások és felszabadítások során a *heap* lyukacsossá válik, a ***coreleft*** függvény leginkább a kiindulási halomméret lekérdezésére szolgál:

- *tiny*, *small* és *medium* modellben:
  ```
  unsigned *coreleft(void);
  ```
- *compact*, *large* és *huge* modellben:
  ```
  unsigned long *coreleft(void);
  ```

A szabványos memóriakezelő függvények a memóriamodellel összhangban levő mutatótípussal (**near** vagy **far**) térnek vissza, amit aztán más könyvtári függvényeknek is átadhatunk argumentumként. A TC 2.0 minden szabványos memóriakezelő függvényhez definiál egy nagyobb "hatótávolságú" megfelelőt. Ezen függvények neveit a szabványos nevek elé helyezett ***far*** szócskával kiegészítve képezték:

```
void far *farmalloc(unsigned long m);
void far *farcalloc(unsigned long n,unsigned long meret);
void far *farrealloc(void far *fptr, unsigned long m);
void farfree(void far *fptr);
unsigned long *farcoreleft(void);
```


Ezek a függvények a *tiny* memóriamodell kivételével minden modell esetén használhatók. A legfőbb jellemzőjük az, hogy egységesen **far** mutatókat használnak, és a lefoglalt memória mérete nem korlátozódik 64 Kbyte-ra. A gondot csak az jelenti, hogy a lefoglalt terület általában nem érhető el a szabványos könyvtári függvényekkel - kivéve azokat az eseteket, amikor 64 Kbyte-nál kisebb blokkot foglalunk **compact**, **large** vagy **huge** modellben lefordított programból. Ezért az ilyen területek feldolgozásáról általában magunknak kell gondoskodni.

A memóriafoglalás a program után a RAM határáig elhelyezkedő területen megy végbe, melynek általában több száz KByte is lehet a mérete. Az alábbi BIGVECT.C példaprogram a *tiny* modell kivételével minden más modellel lefuttatható. A példában 50000 **long** típusú értéknek foglalunk helyet, melyeket aztán a **huge** mutató segítségével végigjárunk:

```
#include <stdio.h>
#include <alloc.h>
#define  NELEM 50000
#define  KCORELEFT (farcoreleft() / 1024)

main()
{
   long huge * ht;
   long index;

   printf("A legnagyobb blokk mérete:%luK\n", KCORELEFT);

                          /* Memóriafoglalás ellenőrzéssel */
   ht = (long huge *) farcalloc(NELEM, sizeof(long));
   if (!ht) {
      printf("\aNincs elegendõ memória!\n");
      return -1;
   }
   printf("A foglalás után: %luK\n", KCORELEFT);
                       /* A lefoglalt terület feltöltése  */
   for (index=0; index<NELEM; index++)
        ht[index]=index;
                     /* A lefoglalt terület felszabadítása */
   farfree(ht);
   printf("A felszabadítás után: %luK\n", KCORELEFT);
}
```

4.7. Speciális könyvtári függvények

4.7.1. Rendezés és keresés

Az alábbi függvények prototípusát az STDLIB.H állomány tartalmazza:

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

Bináris keresés elemek rendezett tömbjében

```
void * bsearch(const void *key, const void *base,
               size_t nelem, size_t width,
               int(*fcomp)(const void *,const void *));
```

A ***bsearch*** függvény a **key* elemet keresi a *base* tömbben. A függvény visszatérési értéke az elsőnek megtalált elemre mutató pointer, vagy NULL, ha nem talált A függvény további paramétereinek jelentése:

nelem	elemek száma a tömbben,
width	egy elem mérete byte-okban kifejezve.
**fcmp*	az összehasonlító függvényre mutató pointer.

Az összehasonlító függvény visszatérési értéke:

```
0 <       ,ha *elem1  < *elem2,
== 0      ,ha *elem1 == *elem2,
> 0       ,ha *elem1  > *elem2.
```

A quicksort algoritmust megvalósító sorbarendező függvény

```
void qsort(void *base, size_t nelem, size_t width,
           int(*fcmp)(const void *, const void *));
```

A ***qsort*** függvény gyorsrendezést hajt végre a *base* mutató által kijelölt tömbben. A paraméterek jelentése megegyezik a ***bsearch*** függvény azonos nevű paramétereinek értelmezésével.

4.7.2. Időfüggvények kezelése

Az idő és a dátum kezeléséhez a TIME.H és/vagy a SYS\TIMEB.H *include* file-okat kell beépíteni a programunkba:

```
#include <time.h>
#include <sys\timeb.h>


char *asctime(const struct tm *tblock);
```

A ***asctime*** függvény 26 karakteres sztringként adja vissza a **tblock* struktúrában tárolt dátumot és időt (a 26-ik karakter az EOS). A sztringet a következő ***asctime*** vagy ***ctime*** hívás felülírja. Példa a sztring felépítésére:

```
Sun Jun 19 04:08:30 1994\n\0


clock_t clock(void);
```

A ***clock*** függvény segítségével két esemény közötti időintervallum határozható meg. Ha az időértékeket másodpercben kívánjuk megkapni, a visszaadott értéket el kell osztani a ***CLK_TCK*** szimbólummal.

```
char *ctime(const time_t *time);
```

A ***ctime*** függvény a dátumot és az időt sztringgé alakítja, és a 26 karakteres sztringre mutató pointerrel tér vissza A **time* értéke egy megelőző ***time*** hívással állítható be.

```
void difftime(time_t time2, time_t time1);
```

A ***difftime*** függvény a megadott két ***time_t*** típusú időpont különbségét határozza meg másodpercekben.

```
void ftime(struct timeb * buf);
```

Az ***ftime*** függvény az aktuális időpontot és dátumot a ***timeb*** típusú struktúrába tölti (**buf*).

```
struct tm *gmtime(const time_t *timer);
```

A ***gmtime*** függvény az aktuális dátumot és az időpontot Greenwich-i idővé konvertálja és egy **struct *tm*** típusú struktúrába tölti.

```
struct tm *localtime(const time_t *timer);
```

A ***localtime*** függvény a helyi dátumot és az időpontot egy **struct *tm*** típusú struktúrába tölti. A ***tm*** struktúra szerkezete:

```
struct tm{
          int tm_sec;
          int tm_min;
          int tm_hour;
          int tm_mday;
          int tm_mon;
          int tm_wday;
          int tm_yday;
          int tm_lsdst;
        };

int *stime(time_t *tp);
```

Az ***stime*** függvény a rendszer időt és a dátumot állítja be. A *tp* pointer mutat az idő értékére, amely 1970. január 1. 00:00:00 óra óta eltelt időt tartalmazza másodpercekben

```
time_t *time(time_t *timer);
```

A ***time*** függvény az aktuális időt állítja be Az időt az 1970 január 1 00:00:00 óra óta eltelt másodpercekben kell megadni.

```
void *tzset( void);
```

A ***tzset*** függvény beállítja a ***daylight***, ***timezone*** és a ***tzname*** globális változókat a ***TZ*** környezeti változók alapján.

4.8. A szöveges képernyő kezelése Turbo C függvényekkel

A Turbo C függvényekben gazdag grafikus könyvtárral rendelkezik, melyek segítségével a teljes képernyőn különféle ábrákat, diagramokat rajzolhatunk fekete/fehérben vagy színesen.
Turbo C-ben írt programok az IBM PC képernyőjét kétfajta módban használhatják

- szöveges módban vagy
- grafikus módban,

ezt a kétfajta módot azonban csak felváltva lehet működtetni. Először ismerkedjünk meg a szöveges móddal, majd nézzük meg részletesen a grafikus mód használatát is.
Mielőtt rátérnénk a szöveges mód ismertetésére, tekintsük át röviden a különféle képernyővezérlő típusokat.

4.8.1. Képernyővezérlő típusok

A PC-hez különféle képernyővezérlőket csatlakoztathatunk. Ezek rendelkezhetnek monochrome, fekete/fehér vagy színes képernyővel és különféle felbontással.
A leggyakrabban használt típusok:

Hercules Monochrome Graphics Adapter	egyszínű: narancssárga, zöld, zöld, vagy szürke kivitelben
VGA Monochrome	papír fehér színű,
Color Graphics Adapter (CGA)	színes (max. 4 szín)
Enhanced Graphics Adapter (EGA)	színes (max 16 szín)
Video Graphics Adapter (VGA)	színes (max. 16 szín)

A vezérlők többsége különböző üzemmódokban használhatók, amelyeket a programból is beállíthatunk.

4.8.2. A szöveges mód képernyőablaka

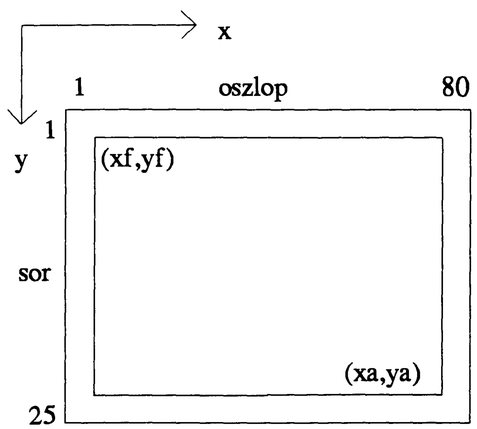
A számítógép a képernyőt alapértelmezés szerint szöveges módban használja. A képernyőablak (*window*) négyzethálóhoz hasonlóan sorokból és oszlopokból áll. Az alapértelmezés szerinti ablak 25 sort és 80 oszlopot tartalmaz. Ez azt jelenti, hogy 80 karaktert írhatunk egy sorba, a 80 feletti karakterek átkerülnek a következő sorba. Ha viszont 25 sornál többet írunk, akkor a 25. sor után amennyi sor megjelenik a képernyő alján, annyi sor tűnik el a képernyő tetejéről. A képernyőből kilépő sorokat nem lehet később megtekinteni, mondhatjuk úgy is, hogy az információ kiszaladt a képernyőből (görgetés - *scroll*). A program írójának kell gondoskodnia arról, hogy a programeredmény képernyője kiértékelhető legyen. Ahhoz, hogy az eredmény színesen és irányítottan adott oszlopba és sorba kerüljön, ajánlott beépítenünk a

```
#include <conio.h>
```

file-t, hogy használhassuk a szöveges mód kezelésére szolgáló függvényeket.

A szöveges képernyőn aktív ablakot definiáló függvény:

```
void window(int xf, int yf, int xa, int ya);
```



4 10 ábra window(1,1,80,25)

Az *xf,yf* az ablak bal felső sarkának, az *xa,ya* pedig a jobb alsó sarkának a koordinátái. Ha az alapértelmezés szerinti ablakot használjuk, akkor nem kell

a függvényt hívnunk, mert `window(1,1,80,25)` a teljes képernyőablakot jelenti. A 4.10. ábrán látható, hogy a képernyő x irányú koordinátái az oszlopokat, az y irányú koordinátái pedig a sorokat jelentik.

Az alapértelmezés szerinti ablakméret, illetve (80x25) mellett használható a 40x25, EGA monitor esetén 80/40x43, VGA monitor esetén 80/40x50 is. Például:

```
window(1,1,80,25);      window(1,1,40,25);
window(1,1,80,43);      window(1,1,40,43);
```

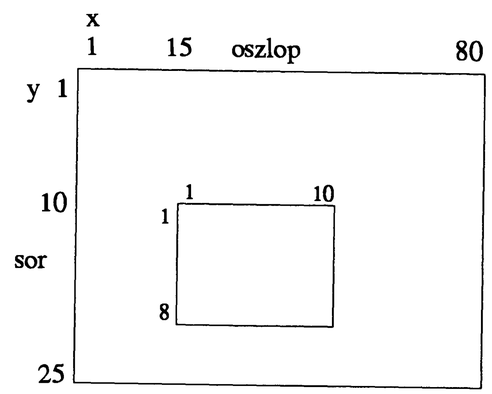
VGA monitornál: `window(1,1,80,50); window(1,1,40,50);` is lehet.
(Ehhez azonban a képernyővezérlőt előzőleg a kívánt módba kell állítanunk)

Ablakban ablakot is létrehozhatunk. A ***window*** függvény koordinátáit abszolút koordinátákkal kell megadni.

Példaként hozzunk létre 10 oszlopot és 8 sort tartalmazó ablakot a fő ablak (15,10) koordinátapontjában:

```
window(15,10,25,18);
```

A 4.11. ábra mutatja a (15,10) pontban létrejött 10x8 méretű ablakot, melynek a bal felső sarokpontja (1,1) lesz.



4 11 ábra window(15,10,25,18)

A szöveges mód függvényeivel állíthatjuk a háttér és a karakterek színét, illetve adott pozícióra írhatunk. A képernyőn bárhol definiálhatunk ablakot, sort írhatunk, törölhetünk és szúrhatunk be. A kurzor pozícióját is

lekérdezhetjük. A programon belül a grafikus módot szöveges móddal oda/vissza lehet váltani. Először tekintsük át a szöveges módra vonatkozó információkat, a használható konstansok és függvények rövid leírását.

4.8.3. Programozás szöveges módban

A szöveges módokat váltogathatjuk a programban, ez természetesen függ a képernyővezérlő típusától. Az aktuális szöveges módot a ***textmode*** függvénnyel állíthatjuk be.

A szöveges mód függvényeit öt csoportra oszthatjuk:
- szöveg kiírása és kezelése,
- ablak és mód vezérlése,
- tulajdonság beállítása,
- állapotlekérdezés,
- hangkeltés.

4.8.3.1. Szöveg kiírása és kezelése

Szöveg írása és olvasása:

cprintf formátumozott adatkivitel képernyőre.

```
int cprintf(const char *formatum[, arg1, ... ]);
```

formatum formátum specifikációt tartalmazó sztring
arg1, .. a kiírandó argumentumok.

cputs sztringet ír a képernyőre.

```
int cputs(const char *string);
```

string a kiírandó a '\0' karakterrel lezárt karaktersorozat

putch egy karaktert ír a képernyőre.

```
int putch(int c);
```

c a kiírandó karakter

getche egy karaktert olvas a billentyűzetről és megjeleníti a képernyőn

```
int getche(void);
```

A beolvasott karaktert adja vissza.

getch egy karaktert olvas a billentyűzetről, amely a képernyőn nem jelenik meg.

```
int getch(void);
```

A beolvasott karaktert adja vissza.

cscanf formátumozott olvasás a billentyűzetről

```
int cscanf(char *format[,cim,...]);
```

A sikeresen beolvasott mezők számával tér vissza.

Szöveg és kurzor mozgatása a képernyőn:

clrscr törli az aktív szöveges ablakot.

```
void clrscr(void);
```

Ha a háttér nem volt fekete, akkor a törlés után a háttér felveszi az előzőekben definiált háttérszínt

clreol törli a sort a kurzor pozíciójától a sor végéig

```
void clreol(void);
```

Törlés után a sor a kurzor helyétől a képernyő széléig háttérszínű lesz.

delline törli a kurzort tartalmazó sort.

```
void delline(void);
```

Törli a kurzort tartalmazó sort és az alatta lévő sorok egy sorral feljebb lépnek a képernyőn

gotoxy pozicionálja a kurzort.

```
void gotoxy(int x, int y);
```

x,y a kurzor új pozíciója

A kurzort az aktív ablak adott pozíciójába mozgatja, az x-edik oszlopba és az y-odik sorba. Az ablak bal felső sarokpontja (1,1),

insline beszúr egy üres sort a kurzort tartalmazó sor alá

```
void insline(void);
```

A sorbeszúrás következtében az utolsó sor kilép a képernyőből.

movetext szöveget másol egyik téglalapból másikba.

```
int movetext(int xf, int yf, int xa, int ya,
             int xu, int yu);
```

xf, *yf* ahonnan másol, a téglalap bal felső sarka,
xa, *ya* jobb alsó sarka,
xu, *yu* ahová másol, a téglalap bal felső sarka.

Szövegblokk elmentése, visszatöltése:

gettext szövegblokkot másol a képernyő téglalap alakú területéről a memóriába.

```
int gettext(int xf, int yf, int xa, int ya, void *hova);
```

xf, *yf* a téglalap bal felső sarka,
xa, *ya* a téglalap jobb alsó sarka,
hova a memóriaterületre mutató pointer.

puttext szövegblokkot másol a memóriából a képernyőre.

```
int puttext(int xf, int yf, int xa, int ya, void *honnan);
```

xf, *yf* a téglalap bal felső sarka,
xa, *ya* a téglalap jobb alsó sarka (ahová másol),
honnan a memóriaterületre mutató pointer (ahonnan másol).

Mindkét függvény esetén a koordinátákat abszolút koordinátaértékkel kell megadni. Egy pozíciónak 2 byte felel meg. Sikeres végrehajtás esetén a visszatérési érték 1, különben 0.

4.8.3.2. Ablak létrehozás és üzemmód beállítás

textmode beállítja a képernyőt a kívánt szöveges módba.

```
void textmode(int ujmod);
```

ujmod a szöveges mód típusa.

window egy képernyőablakot definiál

```
void window(int xf, int yf, int xa, int ya);
```

xf, *yf* az ablak bal felső sarka,
xa, *ya* az ablak jobb alsó sarka.

A szöveges ablak minimális mérete 1 oszlop és 1 sor. Az alapértelmezés szerinti a szöveges ablak a teljes képernyő: window(1,1,80,25).

4.8.3.3. Tulajdonságok beállítása

Az előtér (írás) és a háttér színének beállítása:

textcolor beállítja az előtér színét, ez lesz a karakter színe.

```
void textcolor(int ujszin);
```

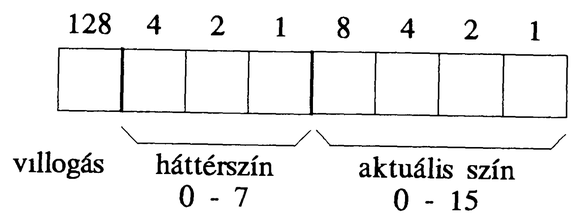
ujszin a karakter színe (0 - 15).

A színkonstans (3 2. táblázat) is megadható. A BLINK hatására a szöveg villog.

textattr beállítja az előtér és a háttér színeit (attribútumát).

```
void textattr(int ujattr);
```

ujattr attribútum
Egyetlen hívással is be lehet állítani a háttér és az írás (előtér) színét.



textbackground beállítja a háttér színét.

```
void textbackground(int ujszin);
```

ujszin a háttér színe.

Az *ujszin* megadható (0-7) számértekkel vagy szimbolikus konstanssal is.

Intenzitás beallítasa:

highvideo magas intenzitás beállítása.

```
void highvideo(void);
```

lowvideo alacsony intenzitás beállítása.

```
void lowvideo(void);
```

normvideo az eredeti, normál intenzitás beállítása.

```
void normvideo(void);
```

4.8.3.4. Működési információk lekérdezése

gettextinfo az aktuális szöveges ablak információjával feltölti a *text_info* struktúrát.

```
void gettextinfo(struct text_info *r);
```

r struktúrába tölti az információkat.

A ***text_info*** struktúra is a CONIO.H file-ban van definiálva:

```
struct text_info{
 unsigned char winleft;
 unsigned char wintop;       /* az ablak bal felső sarkának
                                koordinátája */
 unsigned char winright;
 unsigned char winbottom;    /*az ablak jobb alsó sarkának
                                koordinátája */
 unsigned char attribute;    /* szöveg tulajdonsága */
 unsigned char normattr;     /* normal tulajdonság */
 unsigned char currmode;     /* BW40, BW80, C40 vagy C80 */
 unsigned char screenheight;/* a képernyő magassága */
 unsigned char screenwidth; /* a képernyő szélessége */
 unsigned char curx;
 unsigned char cury;         /* a kurzor aktuális pozíciója */
},
```

wherex megadja a kurzor helyének *x* koordinátáját, amely az aktuális ablakhoz képest relatív távolságot jelent.

```
int wherex(void);
```

A visszatérési érték 1-80 közötti egész szám

wherey megadja a kurzor helyének *y* koordinátáját, amely az aktuális ablakhoz képest relatív távolságot jelent.

```
int wherey(void);
```

A visszatérési érték 1-80 közötti egész szám.

4.8.3.5. Hangkeltés és program futásának felfüggesztése

Az alábbi függvények a dos.h fejléc file-ban vannak definiálva.

delay felfüggeszti a program végrehajtását az adott időtartamra.

```
void delay(unsigned msec);
```

msec a késleltetés ideje század másodpercben egységben

sleep felfüggeszti a program végrehajtását az adott időtartamra.

```
void sleep(unsigned sec);
```

sec a késleltetés ideje másodperc egységben.

sound megszólaltatja a belső hangszórót az adott frekvencián.

```
void sound(unsigned frekvencia);
```

frekvencia a hang frenvenciája Herz egységben.

nosund kikapcsolja a belső hangszórót

```
void nosound(void);
```

A hangkeltés a ***sound***, ***delay*** és ***nosound*** függvények hívásával történik (a ***nosound*** hívása nélkül a hangszóró nem kapcsolódik ki).

4.8.4 A szöveges mód konstansai

A ***textmode*** függvénnyel a képernyővezérlőt valamely lehetséges szöveges módra állíthatjuk be. A ***textmode*** aktiválásával állíthatunk fekete/fehér ill., színes módot a megfelelő ablakmérettel

Néhány példa a hívásra:

`textmode(BW40);`	fekete/fehér, 40 oszlop
`textmode(BW80);`	fekete/fehér, 80 oszlop
`textmode(C40);`	színes, 40 oszlop
`textmode(3);`	színes, 80 oszlop

Szimbolikus konstans	Numerikus érték	Szöveges mód
LASTMODE	-1	előző szöveges 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	monochrome, 80 oszlop

A szöveg színét a ***textcolor,*** a háttér színét a ***textbackground*** függvények állítják.
A színkonstansok:

Szimbolikus konstans	numerikus érték	e - előtér h - háttér	szín
BLACK	0	e+h	fekete
BLUE	1	e+h	kék
GREEN	2	e+h	zöld
CYAN	3	e+h	türkiz
RED	4	e+h	piros
MAGENTA	5	e+h	lila
BROWN	6	e+h	barna
LIGHTGRAY	7	e+h	világosszürke
DARKGRAY	8	e	sötétszürke
LIGHTBLUE	9	e	világoskék
LIGHTGREEN	10	e	világoszöld
LIGHTCYAN	11	e	világos türkiz
LIGHTRED	12	e	világospiros
LIGHTMAGENTA	13	e	világoslila
YELLOW	14	e	sárga
WHITE	15	e	fehér
BLINK	128	e	villogás

A színekből a szöveg írására mindegyiket (0-15) használhatjuk. A háttérre azonban csak 0-tól 7-ig használhatunk szineket. Legyen a háttér színe türkiz, a szöveg színe kék.

A függvények számkonstanssal vagy színkonstanssal is hívhatók:

```
textbackground(3);        vagy
textbackground(CYAN);
textcolor(1);             vagy
textcolor(BLUE);
```

4.8.5. Mintaprogramok a szöveges mód használatára

4.8.5.1. Szöveges ablakok használata

A CONIO1.C program ablakot (*window*) definiál, amelybe a szöveget ***cprintf*** függvénnyel írjuk. A ***delay*** függvényben megadott értékkel felfüggesztjük a program futását. Általában azért teszünk a programba az ilyen jellegű képernyő kimerevítést, hogy a szemünk észrevegye a változást. A változás a gép gyorsasága miatt olyan gyorsan következik be, hogy a képet éppen csak egy villanásra láthatnánk. Ezt a késleltető módszert a tobbi mintaprogramban is alkalmazzuk.

A CONIO1.C program a ***crleol*** függvény használatát mutatja be, amely a kurzor pozíciójától töröl a sor végéig. A ***gotoxy*** függvénnyel változtatjuk a kurzor helyét.

```
#include <conio.h>
#include <dos.h>
void main()
{
  clrscr();
  textbackground(BLACK);
  window(10,10,60,20);
  cprintf("%s","Iras ");
  delay(1000);
  /* az ablak első pozíciója */
  gotoxy(1,1);
  clreol();  /* törlés a kurzortól jobbra */
  delay(1000);

  cprintf("%s","Törlés és felülírás *****");
  delay(1000);

  gotoxy(20,1);
  clreol();
  delay(1000);
  cprintf("%s"," módosítás ");

  getch();
}
```

A CONIO2.C programban a ***delline*** függvénnyel a kurzort tartalmazó sort töröljük és az ***insline*** segítségével sort szúrunk be.

```
#include <stdio.h>
#include <conio.h>
#include <dos.h>
void main()
{
   /* ablakban való görgetés  */
   clrscr();
   textbackground(BLACK);
   window(15,10,60,20);
   gotoxy(1,1); cputs("1. sor");
   gotoxy(1,2); cputs("2. sor");
   gotoxy(1,3); cputs("3. sor");
   gotoxy(1,4); cputs("4. sor");
   gotoxy(1,5); cputs("5. sor");
   gotoxy(1,6); cputs("Utolsó sor");
   delay(2000);

   gotoxy(15,1);  cputs("az ablak elsõ sorának törlése");
   delay(2000);
   delline();
   delay(2000);

   gotoxy(15,1); cputs("az ablak elsõ sorának törlése");
   delay(2000);
   delline();
   delay(2000);

   gotoxy(15,4); cputs("az ablak 4. sorának törlése");
   delay(2000);
   delline();
   delay(2000);

   gotoxy(15,2);  cputs("beszúrás az ablak 2. sorába");
   delay(2000);
   gotoxy(15,2); clreol();
   gotoxy(1,2);
   insline();
   cputs("2. sor: sorbeszúrás");
   getch();
}
```


A CONIO3.C program két ablakot nyit, az ablakokhoz különböző színű hátteret rendel. Értékeljük ki az alábbi programrészletet:

```
#include <conio.h>
#include <dos.h>
void main()
{
   /* teljes képernyő ablaka */
   textbackground(LIGHTGRAY);
   clrscr();

   /* első ablak */
   window(10,10,20,15);
   textbackground(CYAN);
   clrscr();
   textcolor(RED); gotoxy(1,1);
   cputs("1. Sor ");
   delay(2000);
   gotoxy(1,2);
   cputs("2. Sor ");
   delay(2000);
   gotoxy(1,3);
   cputs("3. Sor ");
   delay(2000);
   gotoxy(1,1);
   textcolor(YELLOW+BLINK);
   cputs("Felülír ");
   delay(2000);

   /* második ablak */
   textcolor(BLACK);
   window(15,20,35,22);
   textbackground(GREEN);
   clrscr();
   gotoxy(5,2);
   cputs("Nyomj Szóközt!");
   getch();

   /* az írás színének visszaállítása */
   textcolor(LIGHTGRAY);
   /* a háttér színének visszaállítása */
   textbackground(BLACK);
   window(1,1,80,25);
   clrscr();
}
```

Az alapértelmezés szerinti 80x25 méretű ablak hátterét világosszürkére színezi a ***clrscr*** függvény hívása. Ez a függvény a képernyőt törli, és az attribútum byte-ot a beállított színekkel tölti fel. A fő ablak (10,10)

koordináta-pontjában létrehozunk egy ablakot, amely 10 oszlopot és 5 sort tartalmaz. A háttér színét türkizre állítja be a ***textbackground*** és a ***clrscr*** hívás végzi el az aktív ablak kiszínezését. Az írás színét a ***textcolor*** függvény pirosra állítja be. A kurzor az aktív ablak bal felső sarkában villog, amely az (1,1) koordinátapontot jelenti. A `cputs(" 1. Sor ");` piros színű kiírása ezért az aktív ablak első sorába kerül, majd a kurzor a következő sor elejére áll. Ugyanígy íródik ki a *2 Sor* ill. a *3 Sor* az ablakban. Bizonyos késleltetés után a kurzor a `gotoxy(1,1)` hatására az ablak első pozíciójára áll, az írás színe villogó sárgára vált a

```
textcolor(YELLOW+BLINK);
```

hívás hatására. A `cputs("Felülír");` végrehajtásakor az *'1 Sor'* szöveg valoban felülíródik. Majd a program feketére váltja az írás színét és egy új ablakban kiírja a *"Nyomj Szóközt"* figyelmeztető szöveget. Egy paraméter-nélküli ***getch*** függvény felfüggeszti a program futását, míg meg nem nyomjuk a Szóköz billentyűt.

A program az alábbi utasításokkal fejeződik be:

```
/* az írás színenek visszaállítása */
textcolor(LIGHTGRAY);
/* a háttér színének visszaállítása */
textbackground(BLACK);
window(1,1,80,25);
clrscr();
```

Ezek az utasítások gondoskodnak a fekete háttérről, az ablak visszaállításáról, valamint az írás színéről, különben visszatérve a DOS rendszerbe, a háttér és az írás színe megmarad.

4.8.5.2. Adat beolvasása és ellenőrzése

CONIO4.C program egy egész típusú adat beolvasásán és ellenőrzésén keresztül mutatja be a szöveges mód függvényeinek használatát.

A program az alább feladatot hajtja végre:

- fejlécet ír ki,
- keri az adat beolvasását,
- megvizsgálja, hogy az adat valóban **int** típusú-e,
- ha a beolvasott adat hibás, akkor az adat alatti sorban pirosan villogó *"Hibás adat!"* és sárga színű *"Nyomj Szóközt!"* üzenet jelenik meg. Ilyenkor a *Szóköz* billentyű leütése után újabb adatot olvasunk be.

- Ha a megadott számadat formailag jó, lehetőség van a javítására, ha az *"Akarja javítani (i/n) "* kérdés mellett <i> betűt ütünk, egyébként <n>-et.
  A válasz után az üzenetsor törlődik és a beolvasás elölről kezdődik az új adat ellenőrzésével.

A program képernyője a következőképpen néz ki:

```
Adat beolvasása billentyûzetrõl

Adat:  s
            Hibás adat! Nyomj Szóközt!
Akarja javítani (i/n):
```

A program megtervezése
A program változói:

adat **int**, a jó adatot tartalmazza,
sz **char** tömb, először sztringbe olvasunk az adat ellenőrzése miatt,
hiba **int**, az *str_to_int* függvény ebbe a változóba teszi, hogy az adat megfelel-e a kért adattípusnak (1 - hiba, 0 - jó).
ch **char**, ebbe a változóba olvassa be a ***getch*** függvény az *i* vagy az *n* válaszkaraktert.

A program listájának magyarázata:

Eltérve az alapértelmezéstől a színes 40 oszlopos szöveges módot választjuk ki:

```
textmode(C40);
```

Ahhoz, hogy a teljes képernyő színe türkiz legyen, először beállítjuk a háttérszínt a `textbackground(CYAN)` függvény hívásával, majd töröljük a képernyőt a ***clrscr*** függvénnyel. A `textcolor(RED)` függvény hívásával az írás színét pirosra állítjuk. A `gotoxy(2,5)` a kurzort a képernyő ötödik sorában a második oszlopba helyezi. A

```
cputs("Adat beolvasása billentyûzetrõl")
```

szöveg a kurzor pozíciójától piros színnel jelenik meg.
Két egymásba ágyazott **do while** ciklust alkalmazunk az adat vizsgálatára. A második **do while** ciklusból addig nem lépünk ki, amíg a *str_to_int* függvény *hiba* paramétere nulla nem lesz, vagyis csak akkor, ha a számadat megfelelő formátumú. Az első **do while** ciklus csak akkor tér vissza, ha az

adatbeolvasáskor kapott adátot javítani akarjuk és *i* választ adtunk, különben a program befejezi a működését.

Fehér színnel a 10. sor 2 oszlopában kerül kiírásra az

Adat

szöveg Az adat olvasásához az írás színét sárgára állítjuk. A kurzor mögötti részt a sor végéig feltétel nélkül töröljük (***clreol***). Az adatot a ***gets*** függvénnyel az *sz* karakter tömbbe olvassuk be és az *str_to_int* függvénnyel az *sz* sztringet az *adat* **int** típusú változóba alakítjuk át. Ha a *hiba* változó tartalma nem zérus, akkor az adat hibás. Lekérdezzük a kurzor *x* és *y* koordinátáit a ***wherex*** és **wherey** függvényekkel.

A hibajelzés a következőképpen néz ki:

Az adatbeolvasás alatti sorban, de az adat kezdetétől négy karakterrel jobbra pirosan villogva kerül kiírásra

```
Hibás adat!
```

Egy üres hellyel távolabb pedig sárga színnel jelenik meg a

```
Nyomj Szóközt!
```

szöveg és mindezt 200 ms ideig 100 Hz-es hangjelzés kíséri.

Ha a *Szóköz* billentyűt leütjük, a teljes sor törlődik és a **do while** ciklus addig ismétlődik, amíg a *hiba* változó nem nulla. Nulla visszatérés a hibátlan adat bevitelét jelenti Két sorral lejjebb fekete színnel kerül kiírásra

```
Akarja javítani (i/n):
```

A ***getch*** a *ch* változóba olvassa be a leütött karaktert. Ezt a karaktert az ***toupper*** függvény nagybetűvé alakítja, így elég az *I* és az *N* betűt vizsgálni.

Az *i* válasz hatására az adatbeolvasás és az ellenőrzés elölről kezdődik, *n* válasz esetén a program befejezi a működését.

A CONIO4.C program listája:

```
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#include <dos.h>
int str_to_int(char *s, int *code);
void main()
{
   int        adat, hiba, x, y;
   char       ch, sz[20];
   textmode(C40);
   textbackground(CYAN);
   clrscr();
   textcolor(RED);
   gotoxy(2,5);
   cputs("Adat beolvasása billentyûzetrõl");
   do
   {
      do
      {
        textcolor(WHITE);
        gotoxy(2,10);
        cputs("Adat: ");
        textcolor(YELLOW); clreol();
        x = wherex(); y = wherey();
        gets(sz);
        adat= str_to_int(sz, &hiba);
        if (hiba)
        {
          gotoxy(x+4,y+1);
          textcolor(RED+BLINK);
          cputs("Hibás adat!");
          textcolor(YELLOW);
          cputs(" Nyomj Szóközt!");
          sound(100); delay(200);
          nosound();
          getch();
          gotoxy(2,10); delline();
        }
      }while (hiba);
      do
      {
        gotoxy(2,12);
        textcolor(BLACK);
        cputs("Akarja javítani (i/n): "); ch = getch();
        ch= toupper(ch);
      }while ((ch !='I') && (ch !='N'));
      gotoxy(2,12); delline();
   }while (ch != 'N');
   gotoxy(3,12);
   cprintf("%d",adat);
}
```

```
int str_to_int(char *s,/int *hiba)
{
int i,n,sign;
    *hiba = 0;
    if(strcmp(s,"") == 0) { *hiba = 1; return 0;}
    for(i = 0; s[i] ==' ' || s[i] == '\n' || s[i] == '\t'; i++);
    sign = 1;
    if(s[i] == '+' || s[i] == '-') sign = (s[i++] == '+') ? 1: -1;
    for( n = 0; s[i] != '\0'; i++)
     if (s[i] >= '0' && s[i] <= '9') n = 10*n + s[i] - '0';
     else { *hiba = 1; return 0; }
 return(sign*n);
}
```

A DALLAM.C program a ***sound***, ***nosound*** és a ***delay*** függvények használatát mutatja be egy egyszerű dallam megszólaltatásával.

Ellenőrző kérdések:

1. Mire szolgál a CONIO.H file?
2. Mennyi az oszlopok és a sorok száma az alapértelmezés szerinti ablakban?
3. Melyik függvénnyel hozhatunk létre ablakot?
4. Hol van az aktív ablak (1,1) koordinátapontja?
5. Hangjelzés keltésére milyen függvényeket kell aktiválni?
6. Melyik függvénnyel helyezzük át a kurzor pozícióját?
7. Melyik függvény állítja a háttér színét?
8. Melyik függvénnyel változtathatjuk az írás színét?
9. Melyik függvény törli a képernyőt?
10. Milyen függvényeket kell aktiválnunk, hogy a
    `window(5,5,20,25);`
    ablak háttere kék legyen?
11. Melyik függvénnyel lehet a kurzort tartalmazó sort törölni?
12. Melyik függvénnyel lehet a kurzor pozíciójától a sor végéig törölni?
13. Aktiváljuk a megfelelő függvényt, hogy az írás színe villogó piros legyen!
14. Milyen függvények hívásával állítjuk vissza az alapállapotot, mielőtt visszatérünk az operációs rendszerbe?

Feladatok:

1. Írja ki a billentyűzeten leütött karakter kódját, a programból a <Szóköz> billentyűvel lépjen ki!
   (KODOK.C)

2. Írja ki a teljes IBM kódtáblázatot!
   (IBM_KOD.C)

3. Készítsen a szöveges képernyőn ablakozó technikát használó programot!
   (ABLAKOK.C)

4. Készítsen naptárprogramot, amely az aktuális hónapot, a megelőző és az azt követő hónapot megjeleníti az alábbi módon:
   (NAPTAR.C)

1994 április

H	K	Sz	Cs	P	Sz	V
				1	2	3
4	5	6	7	8	9	10
11	12	13	14	15	16	17
18	19	20	21	22	23	24
25	26	27	28	29	30	

1994 május

H	K	Sz	Cs	P	Sz	V
						1
2	3	4	5	6	7	8
9	10	11	12	13	14	15
16	17	18	19	20	21	22
23	24	25	26	27	28	29
30	31					

1994 június

H	K	Sz	Cs	P	Sz	V
		1	2	3	4	5
6	7	8	9	10	11	12
13	14	15	16	17	18	19
20	21	22	23	24	25	26
27	28	29	30			

5 Tervezzen egy 5x5 méretű számkirakós játékot. A program indításakor bármely billentyű leütésére véletlenszerűen kavarja össze a számokat. A számok tologatása a nyíl billentyűkkel történjen és a játékból bármikor az ESC billentyűvel lehessen kilépni.
(Fazekas Ruck Andrea: TILITOLI.C)

4.9. Grafikus képernyő kezelése Turbo C függvényekkel

A Turbo C GRAPHICS LIB könyvtárában magasszintű- és bitorientált grafikát támogató függvények találhatók, amelyek használatával egyenesek, ívek, körök és más alakzatok rajzolhatók. Rendelkezésre áll továbbá többféle festési minta, vonaltípus és karakterkészlet is.

A grafikus függvények használatához szükséges a TC fejlesztői környezetben az *Options/Linker/Graphics library* opció bekapcsolása, illetve a GRAPHICS.H *include* file megadása, ahol az előredefiniált típusok, konstansok és függvény-deklarációk találhatók. A grafikus könyvtár minden memória modellből egyformán használható, a benne definiált függvények *far* típusúak és *far* mutatókkal paraméterezhetők.

A különféle grafikus kártyák alacsony szintű kezeléséhez szükséges rutinokat a .BGI (*Borland Graphics Interface*) kiterjesztésű file-ok tartalmazzák:

Grafikus meghajtó program	Grafikus hardver
CGA BGI	CGA, MCGA
EGAVGA BGI	EGA, VGA
HERC.BGI	Hercules
ATT.BGI	ATT&T400
PC3270 BGI	3270 PC
IBM8514.BGI	IBM 8514

A négy legelterjedtebb grafikus kártya (CGA, EGA, VGA, Hercules) használatával foglalkozunk.

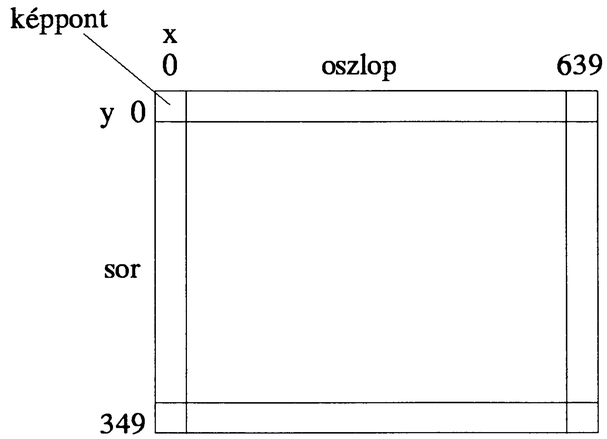
A grafikus rendszer előredefiniált karakterkészleteit a .CHR file-ok tartalmazzák:

GOTH CHR	gót,
SANS CHR	egyszerű,
TRIP.CHR	háromvonalas,
LITT CHR	kisbetű.

4.9.1. A grafikus koordinátarendszer

A rajzok készítéséhez meg kell ismerkednünk a képernyő koordinátarend-szerével. A grafikus rutinok által használt koordinátarendszer origója

(kezdőpontja) a képernyő bal felső sarka. Az *x* koordináta vízszintes irányban balról jobbra haladva, az *y* koordináta pedig lefelé nő. A sorokban és az oszlopokban a pontok száma függ a grafikus kártyától (felbontásától) - ezek értéke a programból a ***getmaxx()*** és a ***getmaxy()*** függvényekkel tudható meg.
Nézzük meg a VGA vezérlő koordinátarendszerét 640x350 pontos felbontás esetén.



4 12 ábra A VGA monitor VGAMED felbontása

Egy (x,y) képernyőpont (*pixel*) helye a képernyőn az x-edik oszlopot és az y-odik sort jelenti.

A grafikus függvényekben a szögeket fokokban kell megadni. A szögek az óramutató járásával ellentétes irányban növekednek. A 3 óra 0 foknak, 12 óra 90 foknak és 9 óra 180 foknak felel meg

4.9.2. Az aktuális pointer (grafikus kurzor)

Grafikus módban, a szöveges mód kurzorához hasonlóan, használható az aktuális mutató (*current pointer* = CP), amely egy képpontra mutat. A szöveges mód kurzora villog a képernyőn, a grafikus kurzor azonban nem látszik. Egyes rutinok mozgatják a grafikus kurzort a képernyőn, mások használják rajzoláskor, kiíráskor. Csak akkor tudunk igazán rajzot tervezni, ha ismerjük ennek a grafikus kurzornak a helyét a képernyőn.

4.9.3 Kiírások a grafikus képernyőn

A karakterkészlet karakterei grafikus módban egy 8*8-as bittérképen definiáltak. A speciális karaktereket tartalmazó karakterkészletek a karaktereket, mint kirajzolandó vonalak halmazát tárolják.

4.9.4. Képernyőlapok és színek

Az EGA, VGA és a Hercules kártyák lehetővé teszik, hogy több képernyőlapot kezeljünk. Különösen jól használható ez a módszer animáció készítéséhez. A színes képernyőkön a rutinok segítségével több színpaletta használható és a palettán lévő színek beállíthatók.

4.9.5. Hibakezelés

Egy speciális függvény, a ***graphresult*** használható arra, hogy a grafikus rutinok hibaüzeneteit lekérdezzük. A ***graphresult*** értéke előredefiniált konstansokkal hasonlítható össze. A ***graphresult*** megadja az utolsó grafikus művelet állapotát, majd ***grOK*** hibaszintet állít be.

4.9.6. A grafikus könyvtár függvényeinek használata

4.9.6.1. A grafikus üzemmód aktivizálása

Az ***initgraph*** függvény segítségével kapcsolható be a grafikus üzemmód. A függvény deklarációja:

```
void far initgraph (int far *graphdriver,
                    int far *graphmode,
                    char far *pathtodriver);
```

A *graphdriver* a használt képernyőtípusnak megfelelő vezérlőt, míg a *graphmode* az adott képernyőn a grafikus módot állítja be. A *pathtodriver* változóban megadható az alkönyvtár, ahol a megfelelő vezérlő állományok (.BGI) keresendők.

A *graphdriver* változó értékei EGA, CGA, Hercules, VGA, ATT, PC3270 és

IBM 8514 kártyák esetén:

```
enum graphics_drivers
  {
   DETECT,                    /* automatikus felderítés */
   CGA, MCGA, EGA, EGA64, EGAMONO, IBM8514, /* 1 - 6 */
   HERCMONO, ATT400, VGA, PC3270,          /* 7 - 10 */
   CURRENT_DRIVER = -1
  };
```

Az ***initgraph*** hívása után lekérdezhető az aktuális grafikus vezérlő neve a ***getdrivername*** függvénnyel:

```
char *far getdrivername(void);
```

A ***graphmode*** lehetséges értékei:

```
enum graphics_modes
  {
    CGAC0       = 0, /* 320x200 paletta 0; 1 lap */
    CGAC1       = 1, /* 320x200 paletta 1; 1 lap */
    CGAC2       = 2, /* 320x200 paletta 2: 1 lap */
    CGAC3       = 3, /* 320x200 paletta 3; 1 lap */
    CGAHI       = 4, /* 640x200              1 lap */
    MCGAC0      = 0, /* 320x200 paletta 0; 1 lap */
    MCGAC1      = 1, /* 320x200 paletta 1; 1 lap */
    MCGAC2      = 2, /* 320x200 paletta 2; 1 lap */
    MCGAC3      = 3, /* 320x200 paletta 3; 1 lap */
    MCGAMED     = 4, /* 640x200              1 lap */
    MCGAHI      = 5, /* 640x480              1 lap */
    EGALO       = 0, /* 640x200 16 szín      4 lap */
    EGAHI       = 1, /* 640x350 16 szín      2 lap */
    EGA64LO     = 0, /* 640x200 16 szín      1 lap */
    EGA64HI     = 1, /* 640x350 4 szín       1 lap */
    EGAMONOHI   = 0, /* 640x350 64K/256K - 1/4 lap */
    HERCMONOHI  = 0, /* 720x348              2 lap */
    ATT400C0    = 0, /* 320x200 paletta 0; 1 lap */
    ATT400C1    = 1, /* 320x200 paletta 1; 1 lap */
    ATT400C2    = 2, /* 320x200 paletta 2; 1 lap */
    ATT400C3    = 3, /* 320x200 paletta 3; 1 lap */
    ATT400MED   = 4, /* 640x200              1 lap*/
    ATT400HI    = 5, /* 640x400              1 lap */
    VGALO       = 0, /* 640x200 16 szín      4 lap */
    VGAMED      = 1, /* 640x350 16 szín      2 lap */
    VGAHI       = 2, /* 640x480 16 szín      1 lap */
    PC3270HI    = 0, /* 720x350              1 lap */
    IBM8514LO   = 0, /* 640x480 256 szín */
    IBM8514HI   = 1  /*1024x768 256 szín */
}
```

Az ***initgraph*** hívása után lekérdezhető a *mode_number*-rel azonosított grafikus mód neve:

```
char *far getmodename(int mode_number);
```

A fenti konstansok használhatók a ***setgraphmode*** függvénnyel, amely alkalmas arra, hogy az adott képernyőtípus egy másik üzemmódjára térjünk át, a képernyő törlése mellett:

```
void far setgraphmode(int mode);
```

Az adott grafikus mód is lekérdezhető a

```
int far getgraphmode(void);
```

függvénnyel.

Nézzünk néhány példát különböző típusú vezérlők valamely grafikus módjának a beállítására és a színek használatára.

Hercules vezérlő

A grafikus üzemmód beállítása:

```
Gd =HERCMONO; Gm = HERCMONOHI;
initgraph(&Gd, &Gm,"");
```

A Hercules vezérlőnek nincs más üzemmódja. A 720x348 felbontás azt jelenti, hogy 0-719 az oszlopok, illetve 0-347 a sorok kiválasztásához használható értéktartomány.

A vezérlő mindössze két színt ismer, amelyből az egyik a háttérszín. Mivel a grafikus lapok száma kettő, lehetőség van arra, hogy a megjelenített lap mellett egy másik lapra is rajzoljunk és a lapokat váltogassuk. Ez a módszer animációs rajzprogramnál meggyorsítja a képek mozgatását a képernyőn.

CGA vezérlő

CGA vezérlő esetén 5 fajta üzemmód használható. Négy üzemmódban a 320x200-as felbontású durva grafikában négy színt használhatunk, a 640x200 felbontású finom grafikában pedig csak két színt. A CGA csak egyetlen grafikus lappal rendelkezik.

Nézzük meg részletesebben a CGA durva grafika színkiválasztását. A négy szín közül egy a háttérszín, amely a 16 szín közül bármely lehet. Négy paletta közül választhatunk további 3 színt a rajzoláshoz. A mód kiválasztásával aktiváljuk a palettát.

Grafikus neve	**mód száma**	**színek sorszáma**		
		1	2	3
CGAC0	0	világoszöld	világospiros	sárga
CGAC1	1	világos türkiz	világoslila	fehér
CGAC2	2	zöld	piros	barna
CGAC3	3	türkiz	lila	világosszürke
CGAHI	4			

Például válasszuk ki a CGA monitor CGAC1 (1) palettáját:

```
Gd = CGA; Gm = CGAC1;
initgraph(&Gd, &Gm,"");
```

Válasszuk háttérszínnek a kéket és váltogassuk a rajzolás színét:

```
setbkcolor(BLUE);   háttérszín kék
setcolor(1);        rajzolás színe   világos türkiz,
setcolor(2);                         világoslila,
setcolor(3);                         fehér.
```

EGA vezérlő

Az EGA vezérlő 64 színt kezel, azonban ebből egyidejűleg csak 16 színt használhatunk (paletta).
Válasszuk ki az EGA vezérlő nagyfelbontású üzemmódját:

```
Gd = EGA; Gm = EGAHi;
initgraph(&Gd, &Gm,"");
```

Rajzoljunk kék színnel:

```
setcolor(BLUE);   vagy
setcolor(1);
```

VGA vezérlő

VGA vezérlő a lehetséges 256 színből egyidejűleg csak 16 színt képes megjeleníteni. A VGALO és a VGAMED mód kiválasztása esetén 2 grafikus lapon is dolgozhatunk.

Az adott vezérlő esetén, a használható módok maximális számát adja a

```
int far getmaxmode(void);
```

függvény.
A beállított vezérlőnek megfelelő grafikus módok értékhatára is lekérdezhető a

```
void far getmoderange(int graphdriver,
                      int far *lomode,
                      int far *himode);
```

függvénnyel.
A hardver ellenőrzésével felderíthető a számítógépben található képernyővezérlő is a hozzátartozó legjobb (maximális) felbontást biztosító üzemmód:

```
void far detectgraph(int far *graphdriver,
                     int far *graphmode);
```

függvénnyel.
A grafikus beállítások visszaállíthatók a feltételezett értékekre a

```
void far graphdefaults(void);
```

függvénnyel.

4.9.6.2. Visszatérés a szöveges üzemmódba

A visszatérés szöveges üzemmódba a

```
void far closegraph(void);
```

függvény aktivizálásával történik. A

```
void far restorecrtmode(void);
```

a grafikus képernyő aktiválása előtti szöveges állapotot állítja vissza.

4.9.6.3. Színek használata

A CGA képernyőn az előredefiniált paletták színei használhatók (CGA0, CGA1, CGA2, CGA3). Az EGA, VGA képernyőn definiálhatók a paletta színei az alábbi típus felhasználásával:

```
#define MAXCOLORS 15
    struct  palettetype
    {
     unsigned char size;
     signed char colors[MAXCOLORS+1];
    }
```


A *MAXCOLORS* előredefiniált konstans. A *size* a paletta mérete és a *colors* tömb tartalmazza a színkódokat. A *colors* tömb indexeléséhez az alábbi konstansokat használhatjuk:

```
enum COLORS
        {
           BLACK,           /* fekete */
           BLUE,            /* kék */
           GREEN,           /* zöld */
           CYAN,            /* cián */
           RED,             /* piros */
           MAGENTA,         /* bíbor */
           BROWN,           /* barna */
           LIGHTGRAY,       /* világosszürke */
           DARKGRAY,        /* sötétszürke */
           LIGHTBLUE,       /* világoskék */
           LIGHTGREEN,      /* világoszöld */
           LIGHTCYAN,       /* világoscián */
           LIGHTRED,        /* világospiros */
           LIGHTMAGENTA,    /* világosbíbor */
           YELLOW,          /* sárga */
           WHITE            /* fehér */
        };
```

Az aktuális paletta mérete megegyezik a

```
int far getpalettesize(void);
```

függvény értékével.

Az EGA, VGA paletta színei állíthatók a

```
void far setpalette(int colornum, int color);
```

függvénnyel színenként, vagy egyszerre a

```
void far setallpalette(struct palettetype far *palette);
```

függvény segítségével.
Az aktuális palettabeállítás le is kérdezhető a

```
void far getpalette(struct palettetype far *palette);
```

függvény meghívásával.
Az aktuális hardver alapértelmezés szerinti palettabeállítása szintén lekérdezhető a

```
struct palettetype * far getdefaultpalette(void);
```

függvénnyel.

A használni kívánt rajzszín beállítható az aktuális palettából választva a

```
void far setcolor(int color);
```

függvény aktivizálásával. Ezentúl a rajzolás színe a *color* argumentummal kiválasztott palettaszín.
Az aktuális vezérlő esetén a legnagyobb színkód értékét szolgáltatja a

```
int far getmaxcolor(void);
```

függvény.
A rajzoláskor használt háttérszín szintén beállítható az aktuális paletta felhasználásával:

```
void far setbkcolor(int color);
```

A beállított rajzolási és háttérszín lekérdezhető a

```
int far getcolor(void);
```

illetve a

```
int far getbkcolor(void);
```

függvények segítségével.

4.9.6.4. Rajzolási módok

A képernyőablak használata

Az EGA, VGA és a Hercules képernyőkön több képernyőoldal (lap) használható a rajzolás során. A rajzolás mindig az aktív képernyőoldalra történik. Az aktív képernyőoldal beállítható a

```
void far setactivepage(int page);
```

függvénnyel. (Az oldalak számozása 0-val kezdődik.)

A képernyőn a látható képernyőoldal az aktív oldaltól függetlenül állítható. Ezzel elérhető, hogy a rajzot "háttérben" készítsük és csak amikor kész, akkor jelenítsük meg. A látható képernyőablak kiválasztására szolgáló függvény a

```
void far setvisualpage(int page);
```

4.9.6.5. Viewport beállítások

Az aktuális grafikus vezérlő és üzemmód beállítások által engedélyezett képernyőméretek lekérdezhetők.

A maximális képpontszám x irányban:

```
int far getmaxx(void);
```

a maximális képpontszám y irányban:

```
int far getmaxy(void);
```

A képernyő bal felső sarkának koordinátái (0,0).

A képernyőn kijelölhető egy ablak. Az ablak kijelölése után minden rajzolási művelet "viewport relatív", azaz a koordinátarendszer kezdőpontja az ablak kezdőpontja. A legnagyobb beállítható ablak a teljes képernyő. Megadhatjuk azt is, hogy a rajz folytatódjon-e az ablak határain kívül (*clip==0*) vagy ne (*clip!=0*). Az ablak kijelölésére szolgáló függvény a

```
void far setviewport(int left,  int top,
                     int right, int bottom,
                     int clip);
```

A (*left*, *top*) az ablak bal felső sarkának megfelelő képernyőpont koordinátái, a (*right*, *bottom*) a jobb alsó sarok koordinátái (képernyő relatív koordinátái). A *clip!=0* esetén vág, *clip==0* esetén nem. A rutin aktivizálása után a grafikus kurzor koordinátái (0,0).
A *viewport*-tal kapcsolatos előre definiált típus:

```
struct viewporttype
   {
    int left, top, right, bottom;
    int clip;
   };
```

Az ***initgraph***, ***graphdefaults*** és a ***setgraphmode*** függvények a teljes képernyőt állítják be *viewport*-nak.

Az aktuális beállítások lekérdezhetők a

```
void far getviewsettings(struct viewporttype far*viewport);
```

függvénnyel.

4.9.6.6. Rajzolási módok

Bitműveletek
Vonalhúzás és poligonrajzolás esetén (***drawpoly***, ***line***, ***linerel***, ***lineto*** és ***rectangle***) a

```
void far setwritemode(int mode);
```

függvény hívásával dönthetünk arról, hogy a kirajzolt vonal milyen bitművelettel kerüljön a képernyőre. Ha a *mode==0*, akkor a vonal minden pontja kikerül a megfelelő beállításokkal, függetlenül attól, hogy mi volt a képernyőn. Ha a *mode==1*, akkor a vonal pontjai kizáró-vagy (XOR) logikai művelettel kerülnek a képernyőre. A GRAPHICS.H file-ból felhasználható konstansok a *COPY_PUT* (0), *XOR_PUT* (1).

4.9.6.7. Vonaltípusok

A vonal rajzolásokhoz (***line***, ***lineto***, ***rectangle***, ***drawpoly***, ***arc***) használt vonaltípus beállítható a

```
void far setlinestyle(int linestyle, unsigned upattern,
                      int thickness);
```

függvénnyel. A *linestyle* paraméter lehetséges értékei:

```
enum line_styles
 {
  SOLID_LINE   = 0,     /* folytonos vonal */
  DOTTED_LINE  = 1,     /* pontvonal */
  CENTER_LINE  = 2,     /* középvonal */
  DASHED_LINE  = 3,     /* szaggatott vonal */
  USERBIT_LINE = 4,     /* felhasználó által definiált */
 };
```

A *upattern*-t a rutin figyelmen kívül hagyja, ha a *linestyle!=4*. Ha a *linestyle==4*, akkor az *upattern* egy 16 bites mintát jelent, amelynek ismétlésével épül fel a vonal A vonalvastagság lehetséges értékei:

```
enum line_widths
 {
  NORM_WIDTH  = 1,      /* szimpla vonal */
  THICK_WIDTH = 3,      /* vastag vonal */
 };
```

A GRAPHICS.H definiál egy olyan típust, amely a vonaltípus-beállítás értelmezésére szolgál.

```
struct linesettingstype
{
  int linestyle;
  unsigned upattern;
  int thickness;
};
```


Az aktuális beállítás lekérdezhető a

```
void far getlinesettings(struct linesettingstype
                         far *lineinfo);
```

függvény hívásával.

4.9.6.8. X:Y viszony

Előfordulhat, hogy a képernyőre rajzolt kör ellipszis lesz. Ez a képernyő beszabályozásán kívül állítható szoftverrel is a

```
void far setaspectratio(int xasp, int yasp);
```

függvény segítségével. A feltételezett viszony az aktuális vezérlő maximális *x* és *y* irányú felbontásától függ. Az aktuális beállítás lekérdezhető a

```
void far getaspectratio(int far *xasp, int far *yasp);
```

függvénnyel.

4.9.6.9. Festési módok

A festési mód (***fillpoly***, ***bar***, ***bar3D***, ***pieslice***) a

```
void far setfillstyle(int pattern, int color);
```

függvénnyel állíthatók be. A festés mintáját a *pattern* adja meg. A lehetséges értékek és a használható előredefiniált konstansok:

```
enum fill_patterns
 {
   EMPTY_FILL,          /* háttérszínnel tölt  */
   SOLID_FILL,          /* az adott színnel tölt */
   LINE_FILL,           /* vízszintes */
   LTSLASH_FILL,        /* 45°-os */
   SLASH_FILL,          /* 45°-os vastag */
   BKSLASH_FILL,        /* 135°-os vastag */
   LTBKSLASH_FILL,      /* 135°-os */
   HATCH_FILL,          /* 90°-os keresztsraffozás */
   XHATCH_FILL,         /* 45°-os keresztsraffozás */
   INTERLEAVE_FILL,     /* 45°-os inverz sraffozás */
   WIDE_DOT_FILL,       /* ritkán pöttyöz */
   CLOSE_DOT_FILL,      /* sűrűn pöttyöz */
   USER_FILL            /* felhasználó által definiált */
 };
```

A *color* a lehetséges színbeállításoknak megfelelően választható színkód. A felhasználó is tervezhet festési mintát a

```
void far setfillpattern(char far *upattern, int color);
```

függvénnyel, ahol az *upattern* egy 8 byte-os területre mutat, amely egy 8x8 pontból álló bitmintát definiál. A mintában az 1 értékű bitek világító képpontot jelölnek.

A festéshez egy segédpuffert használ a rendszer, melynek feltételezett mérete 4K és az ***initgraph*** hívásakor automatikusan sor kerül a területfoglalásra a ***_graphgetmem*** függvény segítségével. A ***closegraph*** hívásakor ez a terület felszabadításra kerül. (***_graphfreemem)***
Ha nagy területet festünk (pl. a ***floodfill*** függvény -7 hibát jelez), akkor szükség lehet ennél nagyobb pufferre. A

```
unsigned far setgraphbufsize(unsigned bufsize);
```

függvénnyel helyet foglalhatunk, ahol a *bufsize* a szükséges byte-ok száma. A függvény a hívás előtti puffermérettel tér vissza. A függvényt az ***initgraph*** hívása előtt kell aktivizálni. A festés paraméterei lekérdezhetők a

```
struct fillsettingstype
 {
   int pattern;
   int color;
 };
```

típusú struktúrába a

```
void far getfillsettings(struct fillsettingtype
                         far *fillinfo);
```

függvény felhasználásával. A *pattern* a beállított minta kódja, a *color* pedig a színkód.
Ha a *pattern==USER_FILL*, akkor a

```
void far getfillpattern(char far *pattern);
```

függvénnyel olvashatjuk le a mintát.

4.9.7. Rajzolás a grafikus képernyőn

Több függvény szolgál arra, hogy a grafikus képernyőn a beállításoknak megfelelően rajzoljunk.

4.9.7.1. A grafikus kurzor (aktuális pozíció, tollhegy)

A grafikus függvények többsége használja a grafikus kurzor aktuális pozícióját (***cleardevice***, ***clearviewport***, ***graphdefaults***, ***initgraph***, ***linerel***, ***lineto***, ***outtext***, ***setgraphmode***, ***setviewport***) A grafikus kurzor relatív módon pozícionálható a

```
void far moverel(int dx, int dy);
```

függvénnyel (az aktuális kurzort (*dx,dy*) vektorral eltolva), vagy közvetlen módon az (x,y) koordinátájú pontba mozgatható a

```
void far moveto(int x, int y);
```

függvény meghívásával.
A grafikus kurzor aktuális poziciójának (*x* és *y* koordináta) lekérdezésére használható a

```
int far getx(void);
```

illetve a

```
int far gety(void);
```

Mindkét függvény *viewport*-relatív értéket ad vissza.

4.9.7.2. Pont rajzolás

Az aktuális *viewport* (x,y) koordinátájú pontjába pontot lehet rajzolni a definiált színnel:

```
void far putpixel(int x, int y, int pixelcolor);
```

ahol a színkódot a *pixelcolor* paraméter definiálja. A képernyő bármely (x,y) pontjának színe lekérdezhető az

```
unsigned far getpixel(int x, int y);
```

függvénnyel.

4.9.7.3. Egyenes vonal rajzolása

Egyenes vonalat lehet húzni a képernyőn a

```
void far line(int x1, int y1, int x2, int y2);
```

függvénnyel az (*x1,y1*) pontból az (*x2,y2*) pontig, az aktuális beállításokkal (szín: ***setcolor***, vonaltípus, vonalvastagság: ***setlinestyle***, ***setwritemode***) A függvény nem változtatja az aktuális pointert.

Az aktuális pointertől vonalat lehet húzni relatív módon az adott vektornak megfelelően a

```
void far linerel(int dx, int dy);
```

függvénnyel, az aktuális beállításokkal. Ez a függvény a végpontba helyezi az aktuális pointert. Az aktuális pointertől megadott pontig is lehet vonalat húzni a

```
void far lineto(int x, int y);
```

függvény segítségével. Ez a függvény a megadott pontba (*x,y*) helyezi az aktuális pointert.
Poligon vonal húzható aktuális beállítások mellett a

```
void far drawpoly(int numpoints,
                  int far polypoints[]);
```

függvénnyel. A *numpoints* adja a csúcspontok számát, a *polypoints* egy **int** típusú, *2*numpoints* elemszámú vektor, melynek minden szomszédos elempárja egy-egy csúcspont (x,y) koordinátáit tartalmazza.
Téglalap határvonalait rajzolja meg a

```
void far rectangle(int left,int top,int right, int bottom);
```

függvény.

A téglalap átellenes sarokpontjai (*left, top*) és (*right, bottom*). A függvény szintén a beállított vonal-attribútumokkal dolgozik (***setcolor***, ***setlinestyle***, ***setwritemode***, ***getlinestyle***).

4.9.7.4. Görbe vonalak

Körívet lehet húzni a

```
void far arc(int x, int y,
             int stangle, int endangle, int radius);
```

függvénnyel, az (*x,y*) pont köré *radius* sugárral *stangle* kezdőszögtől *endangle* végszögig. A szögmérés 3 óra iránytól kezdve, az óramutató járásával ellenkező irányú. A szögeket fokban kell megadni.
Teljes kört lehet rajzolni a

```
void far circle(int x, int y, int radius);
```

függvénnyel az (*x,y*) pont köré *radius* sugárral.

Elliptikus ívet lehet húzni a

```
void far ellipse(int x, int y,
                 int stangle, int endangle,
                 int xradius, int yradius);
```

függvénnyel. Az ellipszis középpontjának koordinátái *x,y*. Az ív kezdőszöge az *stangle*, a végszöge az *endangle*. A szögmérés 3 óra iránytól kezdve, az óramutató járásával ellenkező irányú - a szögeket fokban kell megadni. Az ellipszisív x-irányú féltengelye *xradius*, y-irányú féltengelye *yradius*. Mindhárom függvény a beállított vonalhúzás attribútumokat használja (***setcolor***, ***setlinestyle***, ***setwritemode***, ***getlinestyle***).

Az utoljára használt ***arc*** vagy ***ellipse*** függvény beállításai lekérdezhetők a

```
void far getarccoords(struct arccoordstype far *arccoords);
```

függvénnyel. Az *arccoords* mutató a beállított értékeket tartalmazó struktúrára mutat. A **struct** *arccoordstype* előredefiniált típus

```
struct arccoordstype
 {
  int x, y;
  int xstart, ystart, xend, yend;
 };
```

az (*x,y*) a középpont koordinátái, az (*xstart,ystart*) az ív kezdőpontjának koordinátái, az (*xend,yend*) a végpont koordinátái.

4.9.7.5. Festés

A függvények egy része arra szolgál, hogy festett zárt alakzatokat rajzoljunk, illetve zárt alakzatokat befessünk. A festés mintázatát és színét a ***setfillstyle*** és ***setfillpattern*** függvényekkel állíthatjuk be.

Téglalapot rajzolhatunk a képernyőre a beállított festési színnel és mintával (***setfillstyle*** és ***setfillpattern***) a

```
void far bar(int left, int top,
             int right, int bottom);
```

függvénnyel. A téglalap átellenes sarkai (*left,top*) és (*right,bottom*).

Festett háromdimenziós hasábok rajzolhatók a

```
void far bar3d(int left, int top,
               int right, int bottom,
               int depth, int topflag);
```

függvénnyel a beállított színnel és mintával. A hasáb palástlapjának átlópontjai (*left, top*) és (*right, bottom*). A hasáb 45 fokos axonometriában ábrázolt. A mélység a *depth* értékkel állítható, ajánlott értéke (*right-left*)/4. A *topflag!=0* értékénél a fedőlap szintén lefestett. (A *topflag==0* esetén a hasábok egymásra helyezhetők.)

Festett ellipszist lehet rajzolni a beállított színnel és mintával a

```
void far fillellipse(int x, int y,
                     int xradius, int yradius);
```

függvény segítségével. Az (*x,y*) középpont, *xradius* az x-irányú féltengely, az *yradius* pedig *y*-irányú féltengely.
Festett poligon a

```
void far fillpoly(int numpoints, int far *polypoints);
```

függvénnyel rajzolható. A *numpoints* adja a csúcspontok számát, a *polypoints* egy **int** típusú, *2*numpoints* elemszámú vektor, melynek minden szomszédos elempárja egy-egy csúcspont (*x,y*) koordinátáit tartalmazza. A függvény szintén a beállított kifestési adatokkal dolgozik.
A beállított festési adatokkal tetszőleges zárt terület festését teszi lehetővé a

```
void far floodfill(int x, int y, int border)
```

függvény. Az (*x,y*) a *border* színű határokkal rendelkező zárt terület belső pontját definiálja.
Festett körcikk rajzolható

```
void far pieslice(int x, int y,
                  int stangle, int endangle,
                  int radius);
```

függvénnyel, az (*x,y*) pont köré, *radius* sugárral *stangle* kezdőszögtől *endangle* végszögig. A szögmérés 3 óra iránytól kezdve, az óramutató járásával ellentétes irányú és a szögeket fokban kell megadni A függvény a beállított festési adatokkal dolgozik

Színezett ellipsziscikket lehet rajzolni a

```
void far sector(int x, int y,
                int stangle, int endangle,
                int xradius, int yradius);
```

függvénnyel, az (*x,y*) pont köré, *xradius*, *yradius* féltengelyekkel *stangle* kezdőszögtől *endangle* végszögig. A szögmérés 3 óra iránytól kezdve, az óramutató járásával ellenkező irányú - a szögeket fokban kell megadni. A függvény szintén a beállított festési adatokat használja.

4.9.7.6. Törlés a képernyőn

A teljes grafikus képernyő törölhető a

```
void far cleardevice(void);
```

függvény meghívásával. A függvény háttérszínnel törli a képernyőt (***setbkcolor***) és a (0,0) pontba viszi a grafikus kurzort.
Az aktuális *viewport* törölhető a

```
void far clearviewport(void);
```

függvénnyel. A függvény háttérszínnel törli a *viewport*-ot és annak (0,0) pontjába viszi a grafikus kurzort.

4.9.7.7 Bitminták a képernyőn

A képernyőn levő kép valamely téglalap alakú része a memóriába menthető a

```
void far getimage(int left, int top,
                  int right, int bottom,
                  void far *bitmap);
```

függvény segítségével. A elmentendő képterület sarokpontjai (*left, top*) és (*right, bottom*), a *bitmap* pointer jelöli ki a puffert a memóriában. A kimentett terület első két szava a téglalap szélességét és magasságát tartalmazza.
A ***getimage*** függvénnyel elmentett képernyőterület visszatölthető a

```
void far putimage(int left, int top,
                  void far *bitmap, int op);
```

függvény meghívásával. A visszatöltött téglalap kezdő sarokpontja (*left*, *top*), a *bitmap* mutató által definiált puffer tartalmazza a visszatöltendő adatokat. A *op* értéke a visszatöltés módját definiálja az alábbiak szerint:

```
enum putimage_ops
{
  COPY_PUT,    /* másolja*/
  XOR_PUT,     /* XOR */
  OR_PUT,      /* OR */
  AND_PUT,     /* AND */
  NOT_PUT      /* NOT  másolja*/
};
```

A ***getimage*** művelethez szükséges pufferterület méretét adja meg byte-okban az

```
unsigned far imagesize(int left, int top,
                       int right, int bottom);
```

függvény. A puffer max. mérete nagyobb vagy egyenlő 65535 byte (64K-1), ellenkező esetben a visszaadott érték 0xffff (-1).

4.9.8. Szövegek a grafikus képernyőn

Szövegek írhatók a képernyőre különböző karakterkészletekkel, különböző irányokban és méretekkel. Ezek beállíthatók a

```
void far settextstyle(int font,
                      int direction,
                      int charsize);
```

függvénnyel. Különböző karakterkészletek használhatók, melyek közül a *font* paraméter beállítása választja ki az aktuálisat. A *font* paraméter lehetséges értékei:

```
enum font_names
{
  DEFAULT_FONT = 0,      /* 8x8-as karakter */
  TRIPLEX_FONT = 1,      /* triplex karakter */
  SMALL_FONT   = 2,      /* kis karakter */
  SANS_SERIF_FONT = 3,   /* sansserif karakter */
  GOTHIC_FONT  = 4       /* gót karakter */
};
```

A DEFAULT_FONT karakterkészlet karaktereit bittérképek tárolják, míg az összes többi font vonalas (vektoros) tárolású. A *charsize* a rajzolt karakterek méretét adja meg. A DEFAULT_FONT esetén a *charsize=1* jelenti a 8*8-as méretet, a 2 a 16*16-t és így tovább. Minden más karakterkészlet esetén a

charsize=4 felel meg a 8*8-as karakternek. A *direction* választja az írás irányát, az alábbiaknak megfelelően:

```
#define HORIZ_DIR 0 /* az írás balról jobbra történik */
#define VERT_DIR 1  /* az írás felülről lefelé történik */
```

A kiírt szöveg különböző módon igazítható a grafikus kurzorhoz a

```
void far settextjustify(int horiz, int vert);
```

függvény segítségével. A *horiz*, *vert* paraméterek a szöveg vízszintes, illetve függőleges pozíciójának beállítására szolgálnak. Értékük az alábbi konstansok közül kerülhet ki:

```
enum text_just
{
  LEFT_TEXT   = 0,     /* balra       */
  CENTER_TEXT = 1,     /* középre     */
  RIGHT_TEXT  = 2,     /* jobbra      */

  BOTTOM_TEXT = 0,     /* alulra      */
/* CENTER_TEXT = 1,     már definiált */
  TOP_TEXT    = 2      /* felülre     */
};
```

(Az alapértelmezés: *LEFT_TEXT*, *TOP_TEXT*)

A *DEFAULT_FONT* kivételével a karakterek méretei külön beállíthatók a

```
void far setusercharsize(int multx, int divx,
                         int multy, int divy);
```

függvénnyel. A karakter vízszintes méretei a *multx*/*divx*-szeresre, függőleges méreteik a *multy*/*divy*-szeresre változnak a ***settextstyle***-ban beállítotthoz képest.
A szöveg kiírásra vonatkozó beállítások lekérdezhetők a

```
void far gettextsettings(struct textsettingstype
                         far *texttypeinfo);
```

függvény segítségével. A **struct** *textsettingstype* előredefiniált típus szerkezete:

```
struct textsettingstype
{
  int font;
  int direction;
  int charsize;
  int horiz;
  int vert;
};
```

Kiírás előtt a kiírandó szöveg képernyőn elfoglalt méretei szintén lekérdezhetők. A magasságot a

```
int far textheight(char far *textstring);
```

a vízszintes méretet a

```
int far textwidth(char far *textstring);
```

függvények szolgáltatják Mindkét függvény a *textstring* szöveg megjelenítéséhez szükséges képpontok számát adja meg
Szöveg írható a grafikus képernyőre a grafikus kurzor aktuális pozíciójába az

```
void far outtext(char far *textstring);
```

függvénnyel, illetve az

```
void far outtextxy(int x, int y,
                   char far *textstring);
```

függvénnyel az (*x*,*y*) képernyőpozícióra.

4.9.9. Hibakezelés

Minden grafikus művelet a befejezésekor információt szolgáltat a művelet eredményességére vonatkozóan. Ez az információ lekérdezhető a

```
int far graphresult(void);
```

függvénnyel, amely egyben törli a hibainformációt (*grOk*). A visszatérési kódok lehetséges értékei:

```
enum graphics_errors
{
grOk               = 0, /* nincs hiba */
grNoInitGraph      =-1, /* nincs telepítve a BGI */
grNotDetected      =-2, /* nem találja a grafikus hardvert */
grFileNotFound     =-3, /* nem találja a grafikus meghajtót */
grInvalidDriver    =-4, /* nem megfelelő a meghajtó */
grNoLoadMem        =-5, /* nincs elég memória a meghajtónak */
grNoScanMem        =-6, /* nincs elég memória festéskereséshez*/
grNoFloodMem       =-7, /* nincs elég memória festéshez */
grFontNotFound     =-8, /* nincs font file */
grNoFontMem        =-9, /* nincs elég memória fontbetöltéshez*/
grInvalidMode      =-10,/* nem megfelelő a meghajtó üzemmód */
grError            =-11,/* grafikus hiba */
grIOerror          =-12,/* grafikus I/O hiba */
grInvalidFont      =-13,/* nem megfelelő font file */
grInvalidFontNum   =-14,/* nem megfelelő fontszám */
grInvalidDeviceNum =-15,/* nem megfelelő eszközszám */
grInvalidVersion   =-18 /* nem megfelelő verziószám */
};
```


A hibakódnak megfelelő üzenetet adja meg a

```
char *far grapherrormsg(int errorcode);
```

függvény

4.9.10. A grafikus rendszer további lehetőségei

4.9.10.1. Az initgraph függvény működésének módosítása

A memóriafoglalást a grafikus vezérlők (.BGI), fontok (.CHR) és belső pufferek számára a

```
void far *far _graphgetmem(unsigned size);
```

függvény végzi, amely a ***malloc*** függvényt hívja Hasonlóan a memória felszabadítására a

```
void far _graphfreemem(void far *ptr, unsigned size);
```

függvény szolgál, amely a ***free*** függvényt aktiválja.

Saját ***_graphgetmem*** és ***_graphfreemem*** függvények definiálásával a fenti memóriakezelés tetszés szerint módosítható.

4.9.10.2. A grafikus vezérlők és a karakterkészletek beszerkesztése a futtatható programba

1 Lépés:

A BGIOBJ EXE programmal elő kell állítani a beszerkeszthető *object (OBJ)* file-okat. Minden vezérlőhöz és karakterkészlethez tartozik egy alapértelmezés szerinti szimbolikus név:

Vezérlő	**Név**
CGA BGI	CGA_driver
EGAVGA BGI	EGAVGA_driver
HERC BGI	Herc_driver
ATT BGI	ATT_driver
PC3270.BGI	PC3270_driver
IBM8514 BGI	IBM8514_driver
TRIP CHR	triplex_font
LITT CHR	small_font
SANS CHR	sansserif_font
GOTH.CHR	gothic_font

Az alapértelmezés szerinti nevek használatát teszi lehetővé pl.:

small modell esetén
```
BGIOBJ  EGAVGA
BGIOBJ  LITT
```
large modell esetén
```
BGIOBJ /F EGAVGA
BGIOBJ /F LITT
```

2. Lépés:

A kapott *object* file-okat a szerkesztés során kell felhasználni:

- parancssorból : `TCC prg graphics.lib egavga.obj litt.obj`
- TC rendszerből a megfelelő .PRJ file megadásával vagy közvetlenül a GRAPHICS.LIB bővítésével:

```
TLIB graphics + egavga + litt.
```

3. Lépés:

A C programból el kell végezni a beszerkesztett vezérlő karakterkészlet regisztrálását:

```
int registerbgidriver(void (*driver)(void));
```
illetve
```
int registerbgifont(void (*font)(void));
```

A függvények szokásos felhasználása:

```
if (registerbgidriver(EGAVGA_driver) < 0) exit(1);
if (registerbgifont(small_font) < 0) exit(1);
...
initgraph(...);
```

4.9.10.3. Grafikus rendszer bővítése új vezérlőkkel és karakterkészlettel

```
int far installuserdriver(char far *name,
                          int huge (*detect) (void));
```

Az ***installuserdriver*** lehetővé teszi új eszközvezérlő felvételét a belső BGI táblázatba. A *name* paraméter az új .BGI file neve és a *detect* egy olyan függvény mutatója, amely az adott vezérlő esetén elvégzi az automatikus felderítés feladatát. A visszaadott érték az adott vezérlő paraméterszáma. Egyidejűleg maximálisan 10 eszközvezérlő lehet telepítve.
Az

```
int far installuserfont(char far *name);
```


függvény a felhasználó által definiált font file (.CHR) betöltését végzi el. A *name* paraméter a font file nevét tartalmazza. Egyidejűleg max. 20 karakterkészlet használható. A belső fonttáblázat betelését a ***grError*** hibakód jelzi.

4.9.11. A grafikus könyvtár függvényeinek csoportosítása

4.9.11.1. Grafikus rendszer vezérlése

closegraph	lezárja a grafikus rendszert,
detectgraph	ellenőrzi a hardvert, eldönti, hogy milyen grafikus rendszert használjunk, ajánl egy grafikus módot,
graphdefaults	az alapértelmezés szerint értékekkel inicializálja a grafikus rendszer változóit,
_graphfreemem	felszabadítja a memóriát,
_graphgetmem	lefoglalja a memóriát,
getgraphmode	visszatér az aktuális grafikus móddal,
getmoderange	visszatér a specifikált meghajtónak az érvényes legalacsonyabb és legmagasabb módjával,
initgraph	inicializálja a grafikus rendszert, és a hardvert grafikus módba helyezi,
installuserdriver	installálja a felhasználói grafikus meghajtót,
installuserfont	installál egy grafikus betűtípust,
registerbgidriver	regisztrálja a felhasználó által betöltött vagy a programhoz szerkesztett grafikus meghajtót,
restorecrtmode	visszatölti az eredeti, a grafikus mód előtti képernyő-módot,
setgraphbufsize	a belső grafikus puffer méretét határozza meg,
setgraphmode	beállítja a grafikus módot, törli a képernyőt és újratölti az összes grafikus változót az alapértelmezés szerinti értékekkel.

4.9.11.2. Rajzolás és festés

Rajzolás

arc	körívet rajzol,
circle	kört rajzol,
drawpoly	poligon körvonalát rajzolja,
ellipse	ellipszis ívet rajzol,

getarccoords	a körív vagy ellipszis ív utolsó hívásának koordináta-értékeit adja vissza,
getaspectratio	visszatér a grafikus képernyő maximális méretével, amelyből a *pixel* oldalaránya számítható,
getlinesettings	visszatér az aktuális vonal stílusával, mintájával és vastagságával,
line	egyenest rajzol két pont között,
linerel	egyenest húz az aktuális kurzor pozíciótól az adott relatív távolságban lévő pontba,
lineto	egyenest rajzol a grafikus kurzor aktuális pozíciótól a megadott pontba,
moverel	egy relatív távolsággal elmozgatja a grafikus kurzort,
moveto	megadott pontba mozgatja a grafikus kurzort,
rectangle	téglalapot rajzol,
setaspectratio	változtatja a beépített *pixel* oldalarány faktort,
setlinestyle	beállítja az aktuális vonalvastagságot és stílust.

Festés (kitöltés Fill)

bar	rajzol és befest egy téglalapot,
bar3d	rajzol és befest egy téglatestet,
fillellipse	rajzol és befest egy ellipszist,
fillpoly	rajzol és befest egy poligont,
floodfill	befest egy zárt területet,
getfillpattern	visszatér a felhasználó által definiált mintával,
getfillsettings	visszatér az aktuális ecset mintájával és színével,
pieslice	rajzol és befest egy körcikket,
sector	rajzol és befest egy ellipszis cikket,
setfillpattern	kiválasztja a felhasználó által definiált festő mintát,
setfillstyle	kiválasztja a festő mintát és a festő színt.

4.9.11.3. Képernyő, viewport, image és pixel

cleardevice	törli az aktív képernyőt (az aktív lapot),
setactivepage	kijelöli az aktív lapot,
setvisualpage	láthatóvá teszi a megadott grafikus lapot,
clearviewport	törli az aktuális képernyőablakot,
getviewsettings	visszaadja a képernyőablak információit,
setviewport	kijelöli az aktuális képernyőablakot,

getimage	a megadott képmező bittérképét elmenti egy pufferbe,
imagesize	visszatér a tárolni kívánt téglalap alakú kép tárolásához szükséges memóriaterület méretével,
putimage	a korábban tárolt képmező bittérképét a képernyőre helyezi,
getpixel	visszatér a megadott képpont színével,
putpixel	egy pontot rajzol a kijelölt pontba, adott színnel.

4.9.11.4. Szöveg kiírása grafikus módban

gettextsettings	visszatér az aktuális karakterkészlet típusával, irányával, méretével és helyzetével,
outtext	az aktuális pozíciónál szöveget ír,
outtextxy	szöveget ír a megadott pozícióban,
registerbgifont	regisztrál egy beszerkesztett vagy a felhasználó által betöltött betűtípust,
settextjustify	szöveg helyzetének beállítása az ***outtext*** és az ***outtextxy*** függvények számára,
settextstyle	beállítja az aktuális karakterkészletet, kiírásának irányát és a karakterek méretét,
setusercharsize	beállítja a karakter szélesség és magasság méretét,
textheight	megadja a szöveg képpontokban mért magasságát,
textwidth	megadja a szöveg képpontokban mért szélességét.

4.9.11.5. Színvezérlés

Színinformáció lekérdezése

getbkcolor	visszatér a háttér aktuális színével,
getcolor	visszatér az aktuális rajzolási színnel,
getdefaultpalette	a paletta struktúrával tér vissza,
getmaxcolor	a használható színek maximális számával tér vissza,
getpalette	visszatér az aktuális palettával és a paletta méretével,
getpalettesize	visszatér a paletta színeinek számával.

Színbeállítás

setallpalette	a megadott színekkel lecseréli a paletta színeit,
setbkcolor	beállítja az aktuális háttérszínt,
setcolor	beállítja az aktuális rajzolási színt,

setpalette mégváltoztat egy paletta színt.

4.9.11.6. Hibakezelés grafikus módban

grapherrormsg a ***graphresult*** által visszaadott hibakódnak megfelelő üzenettel tér vissza,
graphresult az utoljára végrehajtott grafikus művelet hibakódját adja vissza.

4.9.11.7. Állapot lekérdezése

getarccoords az utoljára rajzolt körív vagy ellipszis ív koordinátáinak értékével tér vissza,
getaspectratio visszatér a grafikus *pixel* oldalarányával,
getbkcolor megadja az aktuális háttérszínt,
getcolor visszatér az aktuális rajz színével,
getdrivername megadja az aktuális grafikus meghajtó nevét,
getfillpattern visszatér a felhasználó által definiált festő mintával,
getfillsettings visszatér az aktuális festő mintával és színnel,
getgraphmode megadja az aktuális grafikus módot,
getlinesettings visszatér az aktuális vonal típusával, mintájával és vastagságával,
getmaxcolor visszatér a használható színek maximális számával,
getmaxmode visszatér az aktuális meghajtó grafikus üzemmódjainak maximális számával,
getmaxx visszatér a pontok maximális számával x irányban,
getmaxy visszatér a pontok maximális számával y irányban,
getmodename visszatér az aktuális mód nevével,
getmoderange visszatér az adott meghajtó üzemmódjainak értékével,
getpalette visszatér az aktuális palettával és annak méretével,
getpixel visszatér az adott képpont színével,
gettextsettings visszatér az aktuális karakterkészlettel, irányával, méretével és helyzetével,
getviewsettings információt ad az aktuális képernyőablakról,
getx visszatér az aktuális pozíció x koordinátaértékével,
gety visszatér az aktuális pozíció y koordinátaértékével.

A grafikus függvények paraméterezését az F3. függelékben foglaltuk össze.

4.9.12. Grafikus programok készítése

4.9.12.1. Téglalap rajzolása

A grafikus programok készítéséhez ismernünk kell a grafikus vezérlőket, a rendelkezésre álló módokat, amelyek meghatározzák a felbontást, a színek számát és a lapok számát. Ezekre az adatokra szükségünk van, hogy megtervezhessük az ismert felbontás alapján a rajzot. A grafikus meghajtóra névvel és konstanssal is hivatkozhatunk, ugyanígy történik a mód kijelölése is. Az oszlop és a sorok száma adja a felbontást.

Első feladatként rajzoljunk egy téglalapot. A GRAFIKA1.C program bemutatja a feladathoz szükséges grafikus függvények és a szükséges fejléc file-ok használatát.
Az alábbi file-okat kell megadni a program elején:

- rajzfüggvényekhez a
  `#include <graphics.h>`
- a ***clrscr*** és a ***cprintf*** függvények használata miatt
  `#include <conio.h>`
- az ***exit*** függvény használata miatt
  `#include <stdlib.h>`

Definiálnunk kell három egész típusú változót a:

Gd grafikus vezérlő,
Gm grafikus mód,
Hibakod a grafikus hiba kódjának

kezelésére.

A grafikus módot az ***initgraph*** függvény nyitja meg, amelynek három paramétere van. Hívása az alábbi módon történhet:

```
Gd = DETECT;
initgraph(&Gd, &Gm, "");
```

`Gd = DETECT;` azt jelenti, hogy az ***initgraph*** függvényre bízzuk a vezérlő típusának és a hozzátartozó legnagyobb felbontás módjának kiválasztását, amelyeket a *Gd* és a *Gm* változóban kapunk meg. Az ***initgraph*** harmadik paraméterének a grafikus vezérlő (BGI) file útvonalát kell tartalmaznia.

Három eset lehetséges:

1. "" üres sztring megadása esetén abban az alkönyvtárban van a megfelelő .BGI, ahol a program elhelyezkedik.
2. "C:\\tc\\bgi" megadjuk az útvonal pontos nevét.
3. programból rákérdezünk az útvonalra.

Mindhárom megoldást bemutatjuk a GRAFIKA1.C, a GRAFIKA2.C és a GRAFIKA3.C programokban.

A grafikus mód megnyitás sikerességét ellenőrizhetjük a ***graphresult*** függvény hívásával, mely által visszaadott értéket a *Hibakod* változóba töltjük. Ha a *Hibakod* változó értéke zérus, akkor a megnyitás sikeres volt. Egyébként a ***grapherrormsg*** függvény *Hibakod* paraméterrel való hívásával a ***cprintf*** függvénnyel szövegesen írathatjuk ki a hiba okát. Például, ha a programunk alkönyvtárában nem találja az EGA vagy VGA vezérlők esetén az EGAVGA.BGI file-t, akkor az alábbi hibajelzést kapjuk:

```
Grafikus hiba: Device driver file not found (EGAVGA.BGI)
```

és a program az exit(1) hívásával befejezi működését.
Ha a grafika bekapcsolása sikeres volt, kezdődhet a rajzolás. A ***rectangle*** függvény hívásához négy paraméter szükséges, a téglalap bal felső *(xf,yf)* és a jobb alsó *(xa,ya)* sarkának koordinátái:

```
rectangle(xf, yf, xa, ya);
```

Ahhoz, hogy a rajz a képernyőn maradjon javasolt a ***getch*** hívása, amely egy billentyű leütéséig várakozik. A grafikát a ***closegraph*** függvény hívásával kell lezárnunk, mielőtt a program a futását befejezné.

A GRAFIKA1.C program listája:

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

void main()
{
 int   Gd, Gm, Hibakod;
   Gd = DETECT;
   initgraph(&Gd, &Gm, "");
   Hibakod = graphresult();
```



```
   if (Hibakod)
   {
     cprintf("Grafikus hiba: %s",
          grapherrormsg(Hibakod));
     exit(1);
   }
   /* rajzolas */
   rectangle(100,100,60,40);
   getch();
   closegraph();
}
```

A GRAFIKA2.C program az *Utvonal* karaktertömbből veszi a .BGI file elérési útvonalát, melyet a "*Kérem az útvonalat* " szöveg kiírása mellett olvas be. Ha a megadott útvonal nem megfelelő, akkor a program ugyanúgy hibát jelez, mint a GRAFIKA1 program tette és befejezi a működését.

```
char Utvonal[20];

clrscr();
cputs("Kérem az útvonalat: "); gets(Utvonal);
initgraph(&Gd, &Gm, Utvonal);
```

Például:

```
Kérem az útvonalat: c:\tc\bgi
```

A GRAFIKA3.C programban az útvonalat beépítettük:

```
#include <string.h>

   char Utvonal[20];

   strcpy(Utvonal, "c:\\tc\\bgi");
```

Az ***strcpy*** másolja be az *Utvonal* karaktertömbbe a .BGI file útvonalát. (Az ***strcpy*** függvény prototípusát a *string h* tartalmazza)

4.9.12.2. Szöveg kiírása grafikus módban

MINTA1.C program minden vezérlőn működik, hiszen fekete/fehér rajzot készít. Ez az egyszerű rajzprogram egy maximális méretű téglalapot rajzol és a téglalap közepére *'Grafikus feladat'* szöveget írja ki. Tervezzük meg a programot!

Három egész típusú változóra lesz szükségünk:

Gd - a grafikus meghajtó,
Gm - a grafikus mód és a
Hibakod - a hibaüzenet számára.

Ha olyan grafikus programot tervezünk, amely automatikusan meghatározza a vezérlő típusát és a legnagyobb felbontásnak megfelelő módot ajánlja fel, akkor a grafikus mód megnyitásánál az ***initgraph*** függvényt a *Gd* paraméterben megadott ***DETECT*** konstanssal kell hívni. Ennek hatására a *Gd* a megfelelő meghajtót és a *Gm* pedig az aktuális meghajtónak megfelelő legnagyobb felbontást jelentő módot tartalmazza.

A ***graphresult*** függvénnyel ellenőrizhetjük, hogy az ***initgraph*** aktiválásával sikerült-e áttérni a grafikus üzemmódra Ha a *Hibakod* változóban visszaadott érték nulla, akkor sikeres volt a grafikus üzemmódba való kapcsolás, ellenkező esetben a ***grapherrormsg*** függvényt a *Hibakod* változóval hívjuk meg, így szöveges formában is kiírathatjuk a hiba okát és a program futása megszakad.

Ha a grafikus módba való kapcsolás sikeres volt, akkor rajzoljunk egy maximális méretű téglalapot a ***rectangle*** függvénnyel. A téglalap bal felső sarka a (0,0) pont és a jobb alsó sarka pedig a grafikus kártyától függően a maximális oszlop, ill. sorok száma legyen, amiket a ***getmaxx*** és a ***getmaxy*** függvényekkel kérdezhetünk le.

A képernyőn a szöveg helyzetét a ***settextjustify*** függvénnyel állíthatjuk be. Ha a ***settextjustify*** mindkét paraméterénél a ***CENTER_TEXT*** konstanst adjuk meg, akkor a szöveg középre kerül. Ezután definiálnunk kell a karakterkészlet típusát, a kiírandó szöveg irányát, valamint a betűk méretét. Jelen példánkban használjuk a normál karakterkészletet, a kiírás iránya legyen vízszintes (***HORIZ_DIR***) és a betűk nagysága 3-szoros

```
settextstyle(DEFAULT_FONT, HORIZ_DIR,3);
```

Ha a szöveget egy adott koordinátapontnál akarjuk megjeleníteni, akkor az ***outtextxy*** függvényt használjuk Az ***outtextxy*** függvény első két paramétere az *x,y* koordinátapont, ahol a szöveg megjelenik, mivel a ***settextjustify*** függvénnyel a szöveget középre centírozzuk, így itt a képernyő közepét kell megadni. A képernyő közepét különböző típusú vezérlők esetén a ***getmaxx*** és a ***getmaxy*** függvények hívásával tudjuk meghatározni. Az ***outtextxy*** függvény harmadik paramétere a kiírandó szöveg, a programban aposztrófok

között, sztringkonstansként adjuk meg a kiírandó szöveget. Gondoskodnunk kell arról, hogy a felrajzolt kép ne tűnjön el azonnal, hanem addig lássuk, amíg akarjuk. Ezt elérhetjük a ***getch*** függvény hívásával, amely billentyű leütésre vár.
A grafikus programot a ***closegraph*** függvény hívásával kell lezárni, amely visszaállítja azt a módot, amely a grafikus üzemmód előtt volt, jelen esetben az eredeti szöveges módot.
A *MINTA1 C* rajzprogram listája a következő:

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

void main()
{
 int   Gd, Gm, Hibakod;

     Gd = DETECT;
     initgraph(&Gd, &Gm, "");
     Hibakod= graphresult();
     if (Hibakod)
     {
       clrscr();
       cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
       exit(1);
     }
     /* rajzolás kezdete */
     rectangle(0, 0, getmaxx(), getmaxy());
     settextjustify(CENTER_TEXT, CENTER_TEXT);
     settextstyle(DEFAULT_FONT, HORIZ_DIR,3);
     outtextxy(getmaxx() / 2, getmaxy() / 2,
          "Grafikus feladat");
     getch();
     /* grafika lezárása */
     closegraph();
}
```

Ha a programot olyan PC-n futtatjuk, amely Hercules, CGA, EGA vagy VGA vezérlővel rendelkezik, akkor a program a vezérlőt felismerve a legnagyobb felbontást választja, megrajzolja a keretet és kiírja a szöveget a képernyő közepére.
Módosítsuk a programunkat úgy, hogy a különböző típusú vezérlők esetén színes módot választunk. A MINTA2.C programban a háttér, a téglalap és a szöveg színesen jelenik meg, kivéve Hercules monitor esetében.

MINTA2.C program listája:

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

void main()
{
 int   szin1,szin2,szin3, Gd, Gm, Hibakod;
    detectgraph(&Gd, &Gm);
    switch( Gd)
    {
     case 1: /* CGA vezérlő  */
          Gm = CGAC1;
          szin1 = 1;
          szin2 = 2;
          szin3 = 3;
          break;
     case 3: /* EGA vezérlő */
          Gm = EGAHI;
          szin1 = LIGHTGRAY;
          szin2 = CYAN;
          szin3 = MAGENTA;
          break;
     case 7: /* Hercules vezérlő */
          Gm = HERCMONOHI;
          szin1 = 1;
          szin2 = 1;
          szin3 = 1;
          break;
    case 9: /* VGA vezérlő */
          Gm= VGALO;
          szin1 = LIGHTGRAY;
          szin2 = CYAN;
          szin3 = MAGENTA;
          break;
    }
     initgraph(&Gd, &Gm, "");
     Hibakod = graphresult();
     if (Hibakod)
     {
       clrscr();
       cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
       exit(1);
     }
     setbkcolor(szin1);
     setcolor(szin2);
     rectangle(0, 0, getmaxx(), getmaxy());
     settextjustify(CENTER_TEXT, CENTER_TEXT);
```



```
     settextstyle(DEFAULT_FONT, HORIZ_DIR,3);
     setcolor(szin3);
     outtextxy(getmaxx() / 2, getmaxy() / 2,
          "Grafika");
     getch();
     closegraph();
}
```

4.9.12.3. A szöveg pozicionálása

A G_SZOVEG.C program az összes karakterkészlettel mintaszöveget ír ki két oszlopban. Az első oszlopban a szövegekre ***LEFT_TEXT***, ***BOTTOM_TEXT***, a másik oszlopban ***LEFT_TEXT***, ***TOP_TEXT*** pozicionálást végeztünk. Hasonlítsuk össze a két kiírást. A futtatáshoz a .CHR file-ok szükségesek.

4.9.12.4. A szöveg szélességének változtatása

Az F_SZOVEG.C a normális, a keskeny és a széles karakterek kiírását mutatja be a ***setusercharsize*** függvény használatával.

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

void main()
{
 int Gd, Gm, Hibakod;
    Gd = DETECT;
    initgraph(&Gd, &Gm,"");
    Hibakod = graphresult();
    if (Hibakod)
    {
       clrscr();
       cprintf("Grafikus hiba: %s ",grapherrormsg(Hibakod));
       exit(1);
    }
    settextstyle(TRIPLEX_FONT,HORIZ_DIR,4);
    outtextxy(50,50,"Normalis");
    setusercharsize(1,2,1,1);
    outtextxy(50,100,"Keskeny");
    setusercharsize(3,1,1,1);
    outtextxy(50,150,"Szeles");
     getch();
    closegraph();
 }
```

4.9.12.5. A meghajtó neyének kiíratása

A MEGHAJTO.C program kiírja a meghajtó nevét a képernyőre. A .BGI file-oknak az aktuális könyvtárban kell lenniük,

```
#include <conio.h>
#include <graphics.h>
#include <string.h>
#include <stdlib.h>

void main()
{
  int  Gd,Gm,Hibakod;
  char *meghajtonev;
  char buff[80];
    Gd =DETECT;
    initgraph(&Gd,&Gm,"");
    Hibakod = graphresult();
    if (Hibakod)
    {
       clrscr();
       cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
       exit(1);
    }
    meghajtonev = getdrivername();
    strcpy(buff,"Meghajtó neve: ");
    strcat(buff,meghajtonev);
    outtextxy(100,100,buff);
    getch();
    closegraph();
}
```

4.9.12.6. A szöveges és a grafikus mód váltása

GRVALTAS.C program mutatja be a grafikából szöveges módba, majd újra grafikába való váltást.

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

void main()
{
 int Gd,Gm, Hibakod;
    Gd = DETECT;
    initgraph(&Gd,&Gm,"");
    Hibakod = graphresult();
```



```
    if (Hibakod)
      {
        clrscr();
        cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
        exit(1);
      }
    rectangle(20,20,290,100);
    outtextxy(25,30,"<RETURN> kilépünk a grafikából.");
      getch();
    restorecrtmode();
    gotoxy(10,10);
    cputs(" Szöveges módban vagyunk!");
    gotoxy(10,12);
    cputs("<RETURN> hatására visszatérünk grafikus módba.");
      getch();
    setgraphmode(getgraphmode());
    rectangle(20,20,290,100);
    outtextxy(25,30,"Grafikus módban vagyunk");
    outtextxy(25,30+textheight("H"),
              "<RETURN> lezárjuk a grafikát");
     getch();
    closegraph();
}
```

4.9.12.7. Alakzat mozgatása

A MOZOG.C program egy tárgyat mozgat, EGA vagy VGA vezérlővel rendelkező gépen fut, és bemutatja az animációhoz szükséges lapozási technikát. A *page* függvény váltogatja a lapokat.

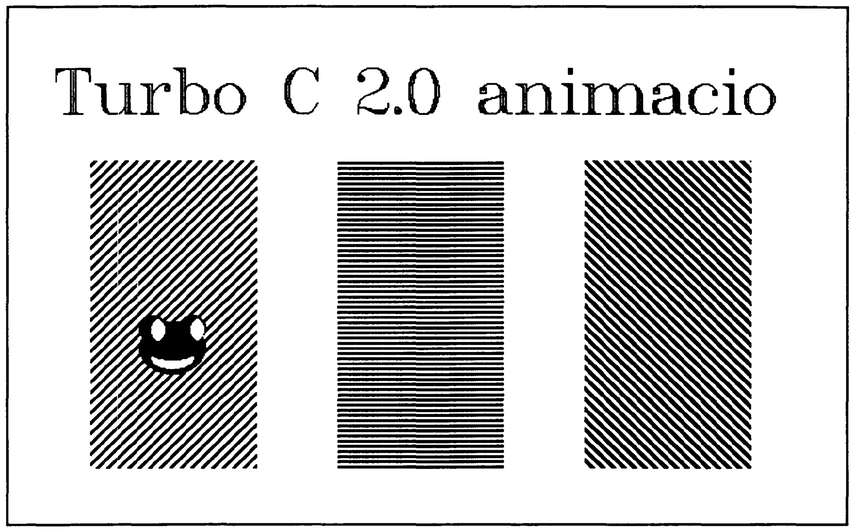
```
#include <conio.h>
#include <graphics.h>
#include <math.h>
#include <stdlib.h>
void page(int *);
void mozog(int *,int,int,int,int,int);

void main()
{
  int Gd, Gm;
  int Active;
  int Hibakod;
  Gd = EGA;      /* vga     */
  Gm = EGAHI;    /* vgamed */
  initgraph(&Gd, &Gm,"");
  Hibakod = graphresult();
```

```
  if (Hibakod)
  {
    clrscr();
    cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
    exit(1);
  }
  cleardevice();
  setvisualpage(0);
  setactivepage(1);
  cleardevice();
  Active = 1;
  setbkcolor(BLUE);
  mozog(&Active,100, 100, 500, 300, 30);
  mozog(&Active,500, 300, 100, 100, 10);
  mozog(&Active,500, 100, 100, 300, 10);
  mozog(&Active,100, 300, 500, 100, 10);
  mozog(&Active,100, 100, 100, 300, 10);
  mozog(&Active,100, 300, 100, 100, 10);
  mozog(&Active,100, 100, 500, 300, 30);
  mozog(&Active,500, 300, 100, 100, 10);
  closegraph();
}

void page(int *Active)
{
    setvisualpage(*Active);
    *Active = 1 - *Active;
    setactivepage(*Active);
    cleardevice();
}

void mozog(int *Active, int X0,int Y0,int X1,int Y1,int V0)
{
  int Vx, Vy, T, X, Y;
  double  Dx, Dy, S0;

    Dx = (double)(X1 - X0); Dy = (double)(Y1 - Y0);
    S0 = sqrt(Dx * Dx + Dy * Dy);
    Vx = (int)(V0 * Dx / S0 + 0.5 );
    Vy = (int)(V0 * Dy / S0 + 0.5);
    T = 0;
    X = X0; Y = Y0;
    do
    {
      setcolor(RED);
      circle(X+2,Y+3,12);
      fillellipse(X, Y, 30,5);
      line(X-10,Y-10,X+2,Y+3);
      line(X+10,Y-10,X+2,Y+3);
```



```
      page(Active);
      X = X0 + T * Vx;
      Y = Y0 + T * Vy;
      T++;
    }while ((abs(T * Vx) < abs(Dx)) && (abs(T * Vy)
            < abs(Dy)));
}
```

A BEKA.C program egyetlen képernyőn tetszőleges képet (békafej) képes mozgatni. A program bemutatja az animációk készítéséhez szükséges műveleteket.



A PALETTA.C program a színskálát körcikkeken mutatja be. ENTER leütésére a kör forogni kezd. (Paletta animáció)

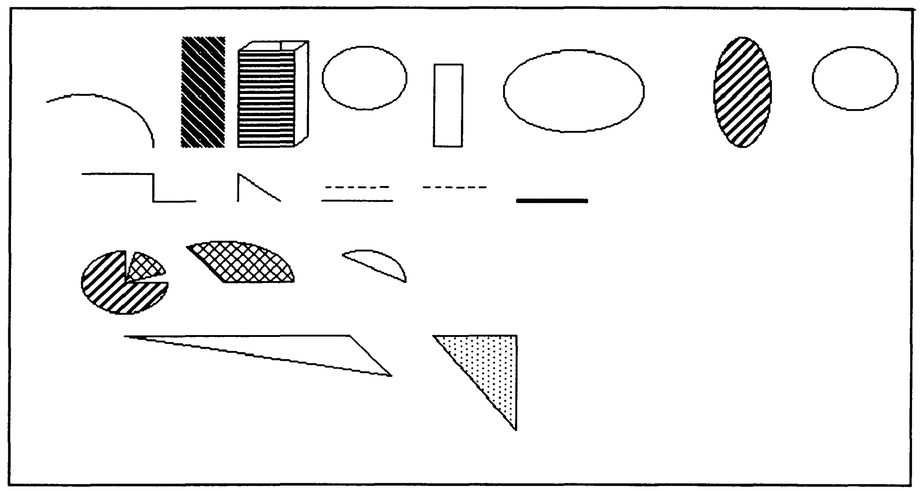
4.9.12.8. Képernyő torzításának kiküszöbölése

Az ASPECT_P.C program EGA és VGA vezérlőkön működik, bemutatja a négyzet és a beleírható kör torzításának számítását.

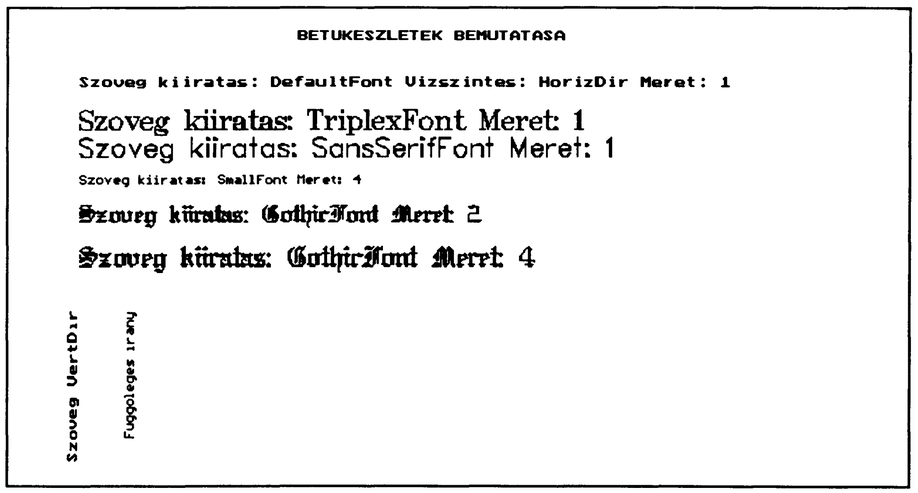
A TORZIT.C program ellipszisből kiindulva addig növeli a torzítási arányt, míg kör nem lesz az ellipszisből.

4.9.12.9. Alakzatok rajzolása

A RAJZ1.C program EGA, VGA képernyőn jeleníti meg az összes rajzolható alakzatot, vonalat, görbeívet, bemutatva az alakzatok festését is.



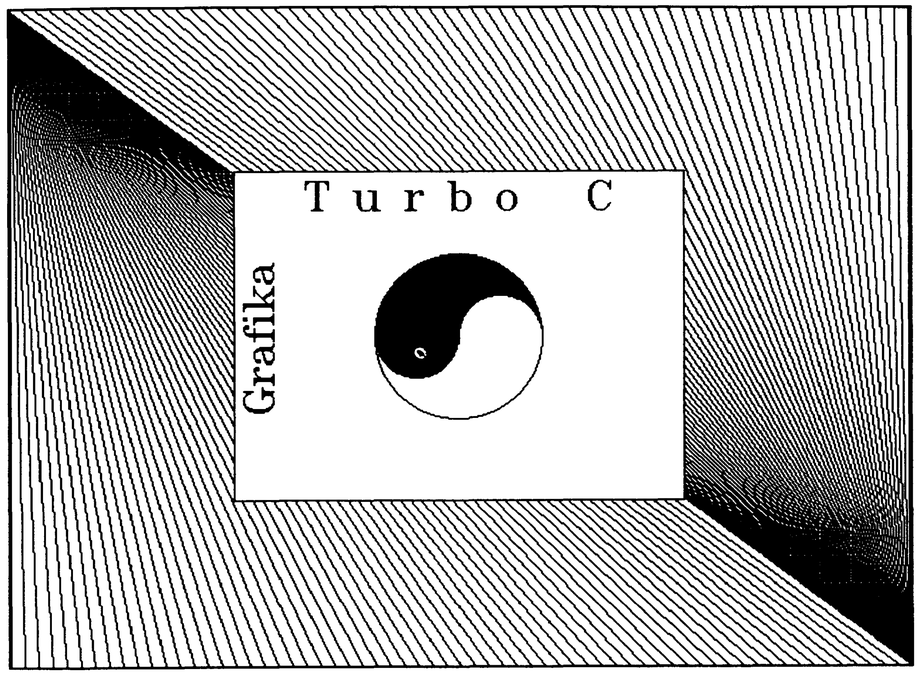
A RAJZ2.C program EGA,VGA képernyőn bemutatja a betűkészleteket, a szöveg nagyságának és irányának programozásával.



A RAJZ3.C program EGA képernyőn működik, és véletlenszám generátorral képzett színkódot, véletlenszám generátorral megadott helyre teszi ki a ***putpixel*** függvény használatával.

A PALETTAD.C program EGA képenyőn mutatja be a paletta átdefiniálását.

A GPLD.C program EGA, VGA képernyőn egy szép grafikus képet hoz létre. 2500 ms ideig kimerevíti a képet, majd a grafikus módot lezárva, visszatér szöveges módba.

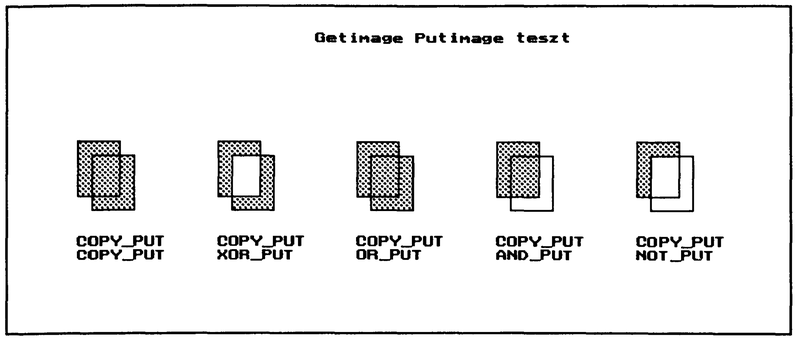


4.9.12.10. Kép kivágása és áthelyezése

A KIVAG.C program a grafikus képernyő (10,20) koordinátapontjánál kivág egy 20x20 képernyőpontú négyzetet, tartalmát a memóriába menti, majd kis idő múlva a (100,100) és a (150,100) koordinátapontoknál megjeleníti.

```
#include <graphics.h>
#include <conio.h>
#include <stdlib.h>
#include <dos.h>

void main()
{
 int  Gd, Gm, Hibakod, meret;
 void *P;
   Gd = DETECT;
   initgraph(&Gd, &Gm, "");
   Hibakod = graphresult();
   if (Hibakod)
    {
      clrscr();
      cprintf("Grafikus hiba: %s",grapherrormsg(Hibakod));
      exit(1);
    }
   /* A teljes képernyőt kifesti */
   bar(0,0,getmaxx(),getmaxy());
   /* Lekéri a kivágandó négyzet méretét */
   setcolor(BLACK);
   rectangle(10,20,30,40);
   meret = imagesize(10,20,30,40);
   P = malloc(meret); /* memóriát foglal le  */
   /* Elteszi a memóriába */
   getimage(10,20,30,40,P);
   delay(3000);
   /* törli a képernyőt */
   cleardevice();
   /* Kiteszi a tárolt négyzetet  */
   putimage(100,100,P, COPY_PUT);
   putimage(150,100,P, COPY_PUT);
     getch();
   closegraph();
}
```


A V_FEKETE.C program minden vezérlőn bemutatja a ***getimage***, ***putimage*** használatát, a képernyőn képezhető összes művelettel.



A V_SZINES.C program EGA, VGA vezérlőn színesen mutatja be a ***getimage*** és a ***putimage*** használatát.

4.9.12.11. Ferdehajítás grafikus ábrázolása

A FERDEHAJ.C program a ferdehajítást szimulálja. A szöveges módban 40 karakterre állítja be a képernyőt. Párbeszédes üzemmódban kérdezi meg a dobás távolságát, melyet el kell találni, majd a dobás szögét és sebességét. Az adatokat ellenőrzi.

Ebbe a programba építettük be az EGAVGA.BGI file-t. A beépítéshez a BINOBJ programmal lefordítottuk a meghajtó szofvert:

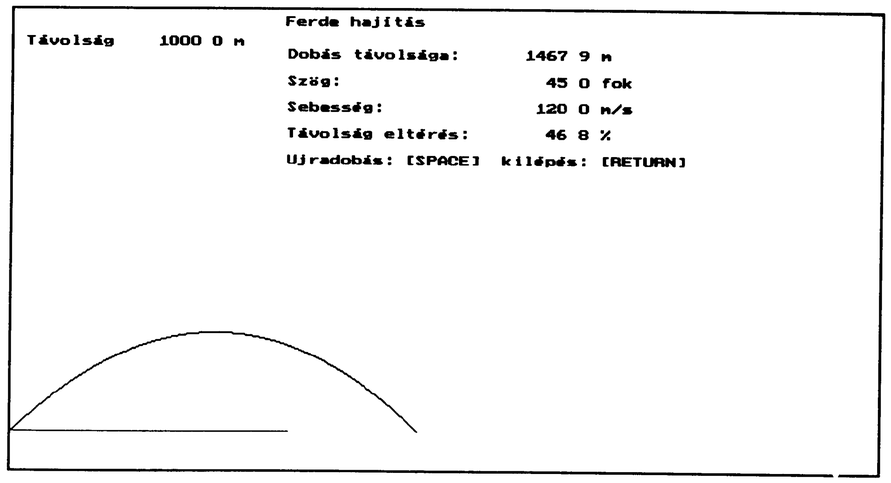
```
BGIOBJ  EGAVGA
```

A keletkezett egavga.obj a szerkesztésnél beépül a .EXE file-ba, amikor az alapértelmezés szerinti EGAVGA_driver néven hivatkozunk rá a programban az alábbi utasítással:

```
if (registerbgidriver(EGAVGA_driver) < 0) exit(1);
```

A FERDEHAJ.EXE futtatásához már nem szükséges külön a grafikus meghajtó. A grafikus meghajtó és a .PRJ file fordító programonként változik.

A FERDETC alkönyvtárban TC (TURBO C) fordítóval kell a FERDEHAJ.PRJ project file-t betölteni, míg a FERDEBC alkönyvtárban a BC (BORLAND C) fordítóval kell a FERDEBC.PRJ file-t betölteni és futtatni. Az alkönyvtárak tartalmazzák a megfelelő EGAVGA.OBJ file-t.



4.9.12.12. Grafikus kurzor mozgatása

A CUR_PROG.C program bemutatja, hogyan lehet grafikus kurzort mozgatni a képernyőn.

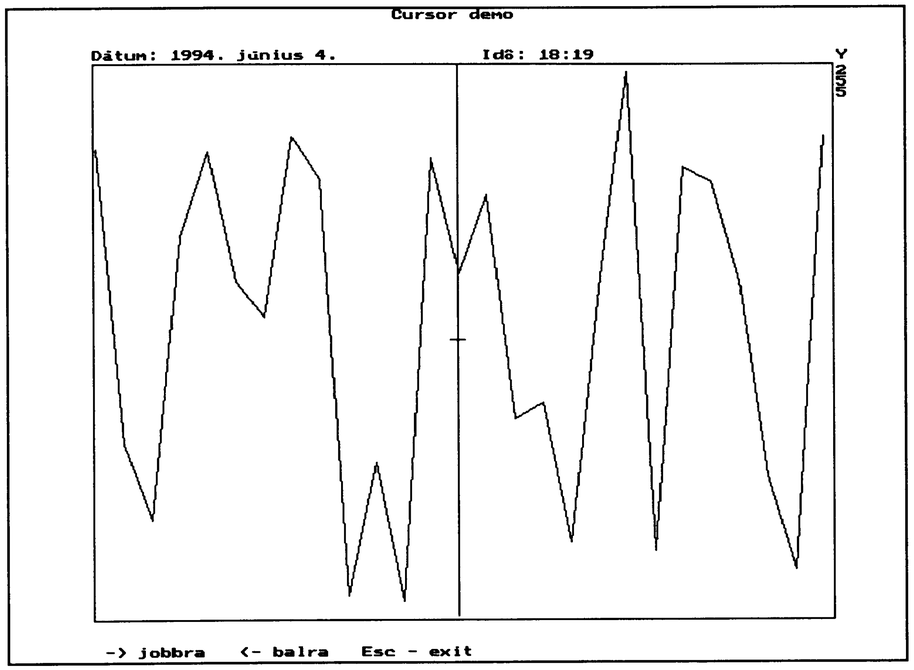
A program EGA és VGA képernyőt tud kezelni. Kirajzolja a képernyőre az aktuális dátumot, a napot valamint az időt is kijelzi.

A program függvényei:

EGA_color	EGA és VGA képernyőre a rajzolás színeit állítja be,
frame_ini	a rajzkeret adatait és a kurzor x irányú mozgatásának nagyságát állítja,
frame	megrajzolja a keretet a fejlécekkel, aktiválja a rajzol függvényt,
adat_ki	a keret jobb szélén függőlegesen írja vissza a kurzor alatti adat értékét,
rajzol	adatokat generál és rajzol,
draw_cursor	mozgatja a kurzort és leolvassa az ábrázolt adatokat, a kurzor jobbra mozgatásánál szaggatott vonallal, balra mozgatásánál telt vonallal jelenik meg Az adat az Y koordináta alatt íródik ki, a kiírás színe is változik.

A grafikusan mozgatható kurzor függvényekkel a kurzor hossza, színe és stílusa (teljes vagy szaggatott vonalú) is beállítható.

A CursoP globális pointerváltozó kurzor képének memóriabeli címére mutat.



A *putcursor* függvény kihelyezi a kurzort a képernyőre, a paraméterei:

h		a kurzor x koordinátája,
v		a kurzor y koordinátája,
hl		a kurzor magasságának mérete, szélessége 5 raszter szimmetrikusan,
vonal	1	a kurzor teljes vonal,
	0	a kurzor szaggatott vonal,
color		a kurzor színe.

A *delcursor* függvény törli a kurzort a képernyőről, a paraméterei:

h	a kurzor x koordinátája,
v	a kurzor y koordinátája,
hl	a kurzor magassága.

Ellenőrző kérdések:

1. Milyen módokban használhatja a Turbo C program a PC képernyőjét?
2. Melyik fejléc file tartalmazza a grafikus függvények prototípusait?
3. Mire szolgál a .BGI file?
4. Mi a kiterjesztése a karakterkészletnek?
5. Melyik függvénnyel kell a grafikus módot megnyitni és mivel zárni?
6. Milyen módszerrel lehet a programba befordítani a grafikus meghajtót?
7. Hol van a grafikus képernyő (0,0) koordinátapontja?
8. Mivel lehet a grafikus képernyőt kimerevíteni, hogy a felrajzolt ábra a képernyőn maradjon?

Feladatok:

1. Írjon programot, amely grafikus módban kirajzolja az olimpiai karikákat. (OLIMPIA.C)
2. Írjon programot, amely a képernyőn adott osztásban négyzethálót rajzol! (RASTER.C)
3. Írjon programot, amely egy házat rajzol ki és a falakat kifesti! (HAZ.C)
4. Írjon évdiagram rajzolására programot! (EV_DIAG.C)
5. Általános oszlopdiagram rajzolására tervezzen programot! (OSZLOP.C)
6. Írjon programot, amely egyszerű analóg-digitál órát rajzol a képernyőre! (ORA.C)
7. Rajzolja ki a színskálát, használja a ***palettetype*** típust! (SZINES.C)
8. Írjon programot, amely egy kockát forgat a térben valamelyik csúcspontja körül! (KOCKA.C)
9. Készítsen látványos színeket és különféle alakzatokat használó grafikus programot! (HAROM.C, KALEID.C)

5. Numerikus módszerek és a C nyelv

A mindennapi életben több műszaki, közgazdasági és természettudományi feladat visszavezethető egy-egy matematikai módszerre, például függvényértékek integrálására, függvények ábrázolásához szükséges interpolációra, nemlineáris egyenletek, illetve lineáris egyenletrendszerek megoldására. Az ilyen számítások módszereivel a numerikus analízis foglalkozik.

Ebben a fejezetben néhány numerikus módszer matematikai hátterét és C nyelven történő megvalósítását ismertetjük. A programrészletek és egy-egy feladat futtatási eredménye az elmélet mélyebb megértését támasztja alá.

A következő numerikus módszereket ismertetjük:

1 Lineáris egyenletrendszerek megoldása témakörben foglalkozunk a Gauss-féle kiküszöbölési eljárással, azon belül a részleges és a teljes főelemkiválasztás módszerével.
Bemutatjuk a lineáris egyenletrendszer LU dekompozícióval való megoldását, amely jól alkalmazható mátrix invertálásra és determináns kiszámításra.

2. Egyismeretlenes nemlineáris egyenlet megoldására több módszer közül az alábbiakkal foglalkozunk:
   - behatárolás intervallum-felezéssel,
   - érintő (Newton-Raphson) módszer,
   - húr módszer,
   - Newton-Raphson és a húr módszer együttes alkalmazása,
   - szelő módszer,
   - a fokozatos közelítés (szukcesszív approximáció) módszere.

3 A függvények közelítésére szolgáló interpolációs módszereket és a két paraméteres regressziót is tárgyaljuk. Az interpolációs módszerek közül az alábbiakat ismertetjük:
   - lineáris interpoláció,
   - Langrange interpoláció,
   - Aitken interpoláció,

A két paraméteres regresszió kiszámítását 10 fajta függvénytípusra ismertetjük.

4. A negyedik témakör a numerikus integrálás vagyis az ú.n. numerikus kvadratúra. Ezen belül foglalkozunk:

   - A Newton-Cotes kvadratúra módszerekkel:
     - a téglalap formula,
     - az egyszerű és az összetett trapéz formula,
     - az érintő formula,
     - az egyszerű és az összetett Simpson formula.
   - Romberg eljárással,
   - Nem ekvidisztáns osztású kvadratúrával.

5.1. Lineáris egyenletrendszer megoldása

Számos probléma megoldása visszavezethető lineáris egyenletrendszer megoldására. Direkt és iteratív módszereket egyaránt alkalmazunk. A következőkben a lineáris egyenletrendszer megoldásának fontosabb módszereit ismertetjük.

A lineáris egyenletrendszer

$$
\begin{aligned}
a_{1,1}x_1 + a_{1,2}x_2 + \ldots + a_{1,n}x_n &= y_1 \\
a_{2,1}x_1 + a_{2,2}x_2 + \ldots + a_{2,n}x_n &= y_2 \\
&\ldots \\
&\ldots \\
&\ldots \\
a_{n,1}x_1 + a_{n,2}x_2 + \ldots + a_{n,n}x_n &= y_n
\end{aligned}
$$

mátrix írásmódban

$$
\begin{bmatrix} a_{1,1} & a_{1,2} & \ldots & a_{1,n} \\ a_{2,1} & a_{2,2} & \ldots & a_{2,n} \\ \ldots & & & \\ a_{n,1} & a_{n,2} & \ldots & a_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \ldots \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ \ldots \\ y_n \end{bmatrix}
$$

$$\mathbf{A}\,\mathbf{x} = \mathbf{y}$$

A feladat az $\mathbf{x}$ vektor meghatározása. Feltételezzük, hogy minden ismeretlennek legalább egy együtthatója nem nulla, és hogy az egyenlet együtthatóinak determinánsa nem nulla. Ilyenkor az egyenletrendszernek létezik megoldása:

$$x_1 \quad x_2 \quad \ldots \quad x_n .$$

Amennyiben az egyenletrendszer determinánsa nulla, akkor az egyenletrendszernek általában nincs megoldása, de előfordulhat, hogy végtelen

sok megoldása létezik. Ebben az esetben egy vagy több ismeretlen értéke szabadon megválasztható, a többi ismeretlen pedig ezek lineáris függvénye. A lineáris egyenletrendszerek megoldása különböző módszerekkel történhet

- kiküszöbölési eljárásssal (direkt módszer),
- fokozatos közelítés módszerével (iteratív módszer).

Az **y** jobboldali vektort helyezzük el az **A** mátrix *n+1*-edik oszlopába. Ha az **A** mátrixot egy oszloppal kiterjesztjük, könnyebb az algoritmus kidolgozása

$$a_{i,n+1} = y_i$$

Az egyenlet ekkor az alábbi alakot veszi fel:

$$\sum_{j=1}^{n} a_{i,j} \quad x_j = a_{i,n+1} \qquad (i = 1, 2, \ldots n)$$

5.1.1. Gauss-féle kiküszöbölési eljárás

Legelterjedtebb módszer a Gauss elimináció. A lineáris egyenletrendszert sorozatos átalakításokkal először felső háromszög mátrixú egyenletrendszerré alakítjuk, melyből sorozatos visszahelyettesítéssel közvetlenül megkaphatjuk a megoldásvektor elemeit:

$$\begin{aligned} a'_{1,1}x_1 \quad a'_{1,2}x_2 \quad \ldots \quad a'_{1,n}x_n &= a'_{1,n+1} \\ a'_{2,2}x_2 \quad \ldots \quad a'_{2,n}x_n &= a'_{2,n+1} \\ \\ a'_{n,n}x_n &= a'_{n,n+1} \end{aligned}$$

Ha sikerül ilyen formára transzformálni az együttható mátrixot, akkor az alsó egyismeretlenes egyenletből kiindulva, sorozatos visszahelyettesítéssel megoldható az egyenletrendszer. Az átalakításoknál ügyelni kell arra, hogy a jobb oldali vektor is transzformálódik. Ha egy egyenletrendszert több jobb oldallal kell megoldani, akkor mindent újra kell számolni.

Gauss elimináció

Az **A** *n*-ed rendű mátrixot *n-1* iterációs lépésnek vetjük alá. Legyen *k* az iterációs index, *i* és *j* pedig a mátrix valamely elemének az indexe. Az iteráció képlete:

$$a_{ij}^{(k)} = a_{ij}^{(k-1)} - \frac{a_{ik}^{(k-1)}}{a_{kk}^{(k-1)}} a_{kj}^{(k-1)}$$

ahol:

$k = 1,2, \ldots, n$
$i = k+1, \ldots n \qquad k \neq n$
$j = k, \ldots n+1 \qquad k \neq n$

Az iteráció *n* lépése után megkapjuk a háromszög mátrixot

$$\begin{aligned} a_{1,1}^{(0)}x_1 + a_{1,2}^{(0)}x_2 + \ldots\ldots\ldots + a_{1,n}^{(0)}x_n &= a_{1,n+1}^{(0)} \\ a_{2,2}^{(1)}x_2 + \ldots\ldots\ldots + a_{2,n}^{(1)}x_n &= a_{2,n+1}^{(1)} \\ a_{3,3}^{(2)}x_3 \ldots\ldots + a_{3,n}^{(2)}x_n &= a_{3,n+1}^{(2)} \\ \ldots \\ \ldots \\ a_{n,n}^{(n-1)}x_n &= a_{n,n+1}^{(n-1)} \end{aligned}$$

Az átrendezés után a főátló alatti elemek nullák lesznek. Visszahelyettesítéssel az utolsó egyenletből kell kiindulni. Probléma akkor van, ha a főáltóbeli elem 0 vagy nagyon kicsi szám. A ismeretlenek meghatározását az n-edik ismeretlen számításával kezdjük.

$$x_n = \frac{a_{n,n+1}^{(n-1)}}{a_{n,n}^{(n-1)}}$$

Majd az ismert gyökök visszahelyettesítésével kapjuk meg az egyenletrendszer további gyökeit:

$$x_{n-1} = \frac{1}{a_{n-1,n-1}^{(n-2)}} \left[a_{n-1,n+1}^{(n-2)} - a_{n-1,n}^{(n-2)} \quad x_n \right] \quad \text{stb.}$$

A GAUSS.C program mutatja be a Gauss módszer alkalmazását lineáris egyenletrendszer megoldására. Nézzünk meg néhány részletet a programból. A három betoldás a bővítés helyét jelenti a Gauss módszer továbbfejlesztése céljából:

```
/* 1. betoldás */
for(k = 1; k <= n; k++)
{
 /* 2. betoldás */
  t = a[k][k];
  if( fabs(t) > eps)
  {
    t = 1.0/t;
   for( i = k; i <= n+1; i++)
     a[k][i] = t*a[k][i];

   if (k != n )
   {
     for( i = k+1; i <= n; i++)
     {
       t = a[i][k];
       for(j = k; j <= n+1; j++)
         a[i][j] = a[i][j] - a[k][j] * t;
     }
   }
  }
  else return 1;
}
```

Az átrendezés után a főátló alatti elemek 0-ák lesznek. Visszahelyettesítésnél az utolsó egyenletből kell kiindulni.

```
for(i = n; i >= 1; i--)
{
  t = a[i][n+1];
  for( k = i+1; k <= n; k++)
      t = t - a[i][k] * a[k][n+1];
  a[i][n+1] = t/a[i][i];
}
/* 3. betoldás */
```

Az *a[i][i]* értékét meg kell vizsgálni az osztás miatt, ha nulla vagy nagyon kicsi szám, akkor a függvény hibával térjen vissza (return 1).

1. Feladat: Az ismertetett módszer bemutatására oldjuk meg az alábbi egyenletrendszert a GAUSS.C programmal.

$$\begin{aligned} 10x_1 + 5x_2 + 2x_3 &= 9 \\ 20x_1 + 12x_2 + 7x_3 &= 22 \\ 30x_1 + 19x_2 + 17x_3 &= 45 \end{aligned}$$

A GAUSS.C program eredménye:

```
Lineáris egyenletrendszer megoldása
         Gauss módszerrel

Elemek szama [max. 10]: 3

a[1,1] = 10
a[1,2] = 5
a[1,3] = 2
Jobb oldal:
b[1] = 9

a[2,1] = 20
a[2,2] = 12
a[2,3] = 7
Jobb oldal:
b[2] = 22

a[3,1] = 30
a[3,2] = 19
a[3,3] = 17
Jobb oldal:
b[3] = 45

Az n+1 oszloppal kibõvitett mátrix

 10.00    5.00    2.00    9.00
 20 00   12.00    7.00   22.00
 30.00   19.00   17.00   45.00

Oszt:  a[1,1] =  10.00

1.  iteráció
  1.00    0.50    0.20    0.90
  0.00    2.00    3.00    4.00
  0.00    4.00   11.00   18.00

Oszt:  a[2,2] =   2.00
```

```
2.  iteráció
  1.00     0.50     0.20     0.90
  0.00     1.00     1.50     2.00
  0.00     0.00     5.00    10.00

Oszt:  a[3,3] =   5.00

A módositott mátrix
  1.00     0.50     0.20     1.00
  0.00     1.00     1.50    -1.00
  0.00     0.00     1.00     2.00

Az egyenlet gyökei

x1 = 1.000000
x2 = -1.000000
x3 = 2.000000
```

Gauss módszere könnyen kiterjeszthető olyan

$$\sum_{j=1}^{n} a_{i,j}x_j = a_{i,n+1} \qquad (i=1,2, \ldots m)$$

alakú egyenletrendszerek esetére is, amelyekben az egyenletek és az ismeretlenek száma különböző. Ha $m>n$, akkor általában nincs megoldás, de kivételesen egy, vagy végtelen sok megoldás is létezhet. Ha $n>m$, akkor általában $n-m$ ismeretlen tetszőleges lehet és a többi ezek lineáris függvénye. Előfordulhat azonban az is, hogy nincs megoldás, vagy $n-m$-nél több ismeretlen értéke választható tetszőleges módon.

Gauss elimináció részleges főelemkiválasztással

Ha az együtthatók különbsége nagy, és a főátlóban levő elem (az osztó) értéke kicsi, a megoldás során jelentős hiba keletkezhet. Jobb eredményt kapunk, ha a i-edik ismeretlent az egyenletrendszernek abból az egyenletéből küszöböljük ki, ahol az ismeretlen együtthatója abszolút értékben a legnagyobb. A módszert részleges főelemkiválasztásnak nevezzük.

Ha nem a főátlóbeli elem volt a legnagyobb, az egyenleteket felcseréljük. Ha az egyenletrendszer megoldható, akkor a részleges főelemkiválasztás nem biztos, hogy a legjobb eredményt adja a kerekítési hibák miatt.

A GAUSS.C programot bővítsük úgy, hogy az abszolutértékben legnagyobb elemmel osszunk:

```
/* 2. betoldás  */
 max = 0.0; j = k;
 for (i = k;  i <= n; i++)
 {
  if( fabs(a[i][k]) > max)
  {
    max = fabs(a[i][k]);
    j = i;
  }
 }
 if( j != k)
 {
   /* Ha nem a főátlóbeli elem volt a legnagyobb */
   /* az egyenleteket felcseréljük */
   for( i = k; i <= n+1; i++)
   {
     t = a[k][i];
     a[k][i] = a[j][i];
     a[j][i] = t;
   }
 }
```

2. Feladat: Oldjuk meg az 1. feladatban megadott egyenletrendszert a GAUSSF.C programmal. Értékeljük ki a főelemkiválasztással készült iteráció részeredményét:

```
 Lineáris egyenletrendszer megoldása
   Gauss módszer: fõelemkiválasztás

Elemek száma [max. 10]: 3

a[1,1] = 10
a[1,2] = 5
a[1,3] = 2
Jobb oldal:
b[1] = 9

a[2,1] = 20
a[2,2] = 12
a[2,3] = 7
Jobb oldal:
b[2] = 22

a[3,1] = 30
a[3,2] = 19
a[3,3] = 17
Jobb oldal:
b[3] = 45
```

```
Az n+1 oszloppal kibővitett mátrix

 10.00     5.00     2.00     9.00
 20.00    12.00     7.00    22.00
 30.00    19.00    17.00    45.00

Oszt: 1. oszlop |max. elemével|:  30.00

1. iteráció
  1.00     0.63     0.57     1.50
  0.00    -0.67    -4.33    -8.00
  0.00    -1.33    -3.67    -6.00

Oszt: 2. oszlop |max. elemével|:  -1.33

2. iteráció
  1.00     0.63     0.57     1.50
  0.00     1.00     2.75     4.50
  0.00     0.00    -2.50    -5.00

Oszt: 3. oszlop |max. elemével|:  -2.50

  1.00     0.63     0.57     1.50
  0.00     1.00     2.75     4.50
  0.00     0.00     1.00     2.00

A módositott mátrix
  1.00     0.63     0.57     1.00
  0.00     1.00     2.75    -1.00
  0.00     0.00     1.00     2.00

Az egyenlet gyökei

x1 = 1.000000
x2 = -1.000000
x3 = 2.00000
```

Gauss elimináció teljes főelemkiválasztással

Még jobb közelítést kapunk, ha az *i*-edik lépésben nem feltétlenül az *i*-edik ismeretlent küszöböljük ki, hanem helyette az összes szóbajöhető elemből választott maximális abszolútértékű elemmel generáljuk az eljárást.

A GAUSST.C program mutatja be a teljes főelemkiválasztás módszerét, amelyet a GAUSS.C program a megfelelő helyen való bővítésével érhetünk el:

```
/* 1. betoldás */
for( i = 1; i <= n; i++)
  sor[i] = i;
/* egyenlet sorrendje azonos a sorindex-szel */
```



```
/* 2. betoldás */
max = 0.0; s = k; m = k;
for(i = k; i <= n; i++)
for(j = k; j <= n; j++)
  if( fabs(a[i][j]) > max)
  {
     oszt = a[i][j];
     max = fabs(oszt);
     s = i; m= j;
  }
if( s != k)
{
/* Ha nem a főátlóbeli elem a legnagyobb */
  for( i = k; i <= n+1; i++)
  {
    t = a[k][i];
    a[k][i] = a[s][i];
    a[s][i] = t;
  }
}
  if (m != k)
  {
   /* Egyenletek sorrendjének vizsgálata */
   for( i = 1; i <= n; i++)
   {
     t = a[i][k];
     a[i][k] = a[i][m];
     a[i][m] = t;
   }
   i = sor[k];
   sor[k] = sor[m];
   sor[m] = i;
  }
```

A megoldás rossz sorrendben lesz, ha az egyenlet oszlopait is felcseréltük. A *sor* tömb tartalmazza a jó sorrendet.

```
/* 3. betoldás */
for(i = 1; i <= n; i++)
  x[sor[i]] = a[i][n+1];

for( i = 1; i <= n; i++)
  a[i][n+1] = x[i];
```

Az eredmények így jó sorrendben lesznek. A főátlóban az 1 és a főátló alatti 0-ák mutatják a megoldás pontosságát.

3. Feladat: Oldjuk meg az 1. feladatban megadott egyenletrendszert a GAUSST.C programmal.

```
Lineáris egyenletrendszer megoldása
Gauss módszer: teljes fõelemkiválasztás

Elemek száma [max. 10]: 3

a[1,1] = 10
a[1,2] = 5
a[1,3] = 2
Jobb oldal:
b[1] = 9

a[2,1] = 20
a[2,2] = 12
a[2,3] = 7
Jobb oldal:
b[2] = 22

a[3,1] = 30
a[3,2] = 19
a[3,3] = 17
Jobb oldal:
b[3] = 45

Az n+1 oszloppal kibõvitett mátrix

 10.00    5.00    2.00    9.00
 20.00   12.00    7.00   22.00
 30.00   19.00   17.00   45.00

Oszt: [3,1] indexû |max. elemmel|:  30.00

1. iteráció
  1 00    0.63    0.57    1.50
  0.00   -0.67   -4.33   -8.00
  0.00   -1.33   -3.67   -6.00

Oszt: [2,3] indexû |max. elemmel|:  -4.33

2. iteráció
  1.00    0.57    0.63    1.50
  0.00    1.00    0.15    1.85
  0.00    0.00   -0.77    0.77

Oszt: [3,3] indexû |max. elemmel|:  -0.77

  1.00    0.57    0.63    1.50
  0.00    1.00    0.15    1.85
  0.00    0.00    1.00   -1.00

Az módositott mátrix
  1.00    0.57    0.63    1.00
  0.00    1.00    0.15   -1.00
  0.00    0.00    1.00    2.00
```



```
Az egyenlet gyökei
x1 = 1.000000
x2 = -1.000000
x3 = 2.000000
```

5.1.2. Gauss-Jordan módszer

A lineáris egyenletrendszerek megoldására szolgál az ún. Gauss-Jordan-módszer is. Ennek alkalmazása során a *k*-adik közelítésben a *k*-adik sor együtthatói az

$$a_{k,j}^{(k)} = \frac{a_{k,j}^{(k-1)}}{a_{k,k}^{(k-1)}} \qquad k = 1,2, \ldots n, \quad j = k+1, \ldots n+1$$

képlettel, míg a *k*-tól különböző *i*-edik sor együtthatói az

$$a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - a_{i,k}^{(k-1)} \quad a_{k,j}^{(k)} \qquad i = 1,2 \ldots n,$$
$$i \neq j \quad \text{nem végezzük el},$$
$$k = i+1, \ldots, n+1$$

Itt is lehet alkalmazni a részleges és a teljes főelemkiválasztást, vagyis az *i*-edik ismeretlent nemcsak az *i+1*-edik, *i+2*-edik, ..., *n*-edik egyenletből, hanem az 1., 2., ..., *i-1*-edik egyenletből is kiküszöböljük és így a kiküszöbölés befejezése után már meg is kapjuk az ismeretleneket.

Oldjuk meg az 1. Feladatot az ismertetett módszerrel:

$$\begin{array}{ll} & 10x_1 + 5x_2 + 2x_3 = 9 \\ (0) & 20x_1 + 12x_2 + 7x_3 = 22 \\ & 30x_1 + 19x_2 + 17x_3 = 45 \end{array}$$

$$\begin{array}{ll} & x_1 + 0,5x_2 + 0,2x_3 = 0,9 \\ (1) & 2x_2 + 3x_3 = 4 \\ & 4x_2 + 11x_3 = 18 \end{array}$$

$$\begin{array}{ll} & x_1 - 0,55x_3 = -0,1 \\ (2) & x_2 + 1,5x_3 = 2 \\ & 5x_3 = -10 \end{array}$$

$$(3)\qquad \begin{aligned} x_1 \qquad\qquad &= 1 \\ x_2 \qquad &= -1 \\ x_3 &= 2 \end{aligned}$$

Eddig feltételeztük, hogy az aritmetikai műveleteket pontosan végezzük el. Ilyenkor a feladat pontos megoldását kapjuk. Azonban léteznek olyan módszerek, amelyek az egzakt megoldás közelített értékének meghatározását tűzik ki célul.

5.1.3. Fokozatos közelítések módszere

Ahhoz, hogy a módszert alkalmazhassuk, az egyenletrendszert

$$x_i = \sum_{j=1}^{n} b_{i,j}x_j + b_{i,n+1} \qquad (i = 1, 2,..., n) \qquad (1)$$

alakra kell hozni. Ha az egyenletrendszer

$$\sum_{j=1}^{n} a_{i,j}x_j = a_{i,n+1} \qquad (i = 1, 2,..., n) \qquad (2)$$

alakú, akkor az egyenletrendszert 0-ra redukáljuk:

$$0 = -\sum_{j=1}^{n} a_{i,j}x_j + a_{i,n+1} \qquad (i = 1, 2,..., n)$$

Ezután minden egyenletet megszorzunk egy nullától különböző számmal, például

$\dfrac{1}{a_{i,i}}$ -vel, ha az $a_{i,i} \neq 0$,

és végül az x_i ismeretlent a kapott egyenletek mind a két oldalához hozzáadjuk.

Az (1) egyenletrendszert úgy próbáljuk megoldani, hogy valamilyen

$$x_1^{(0)}, x_2^{(0)}, \ldots, x_n^{(0)}$$

közelítés, például

$$b_{1,n+1}, b_{2,n+1}, \ldots, b_{n,n+1}$$

felhasználásával kiszámítjuk az

$$x_i^{(k)} = \sum_{j=1}^{n} b_{i,j}x_j^{(k-1)} + b_{i,n+1} \quad (i = 1, 2,..., n, k = 1,2,3, ...) \qquad (3)$$

fokozatos közelítéseket. Ha ezek egy véges határértékhez konvergálnak, akkor az csak (1) megoldás lehet, mivel a (3)-ból határátmenettel (1) adódik. Ezért, ha az eljárás konvergál, akkor elég nagy k esetén (3) elég közel lesz a megoldáshoz.

Mitől függ, hogy a (3) fokozatos közelítések tartanak-e egy véges határértékhez? Ha az (1) egyenletrendszer (2)-ből adódott úgy, hogy annak i-edik egyenletéből fejeztük ki az i-edik ismeretlent, akkor a konvergencia feltétele:

$$|a_{i,i}| > \sum_{\substack{j=1 \\ j\neq i}}^{n} |a_{i,j}| \qquad (i =1, 2, ..., n)$$

Ez azt jelenti, hogy olyan egyenletrendszerek oldhatók meg a fokozatos közelítés módszerével, amelyeknél az együtthatómátrix fődiagonálisában álló elemek dominálnak.

5.1.4. Seidel módszer

A fokozatos közelítések módszere módosítható úgy, hogy $x_i^{(k)}$ kiszámításakor már felhasználjuk az $x_1^{(k)}, \ldots, x_{i-1}^{(k)}$ közelítéseket, vagyis (3) helyett a

$$x_i^{(k)} = \sum_{j=1}^{i-1} b_{i,j}x_j^{(k)} + \sum_{j=i}^{n} b_{i,j}x_j^{(k-1)} + b_{i,n+1} \qquad (i = 1,2, .. n) \qquad (4)$$

képleteket használjuk. Az ezen alapuló eljárást Seidel-módszernek nevezzük.

5.1.5. *Lineáris egyenletrendszer megoldása LU dekompozícióval*

Gauss eliminációval a lineáris egyenletrendszer megoldásánál a jobb oldal is transzformálódik:

$$\mathbf{A' \; x = B'}$$

Ha az egyenletrendszert több jobb oldallal kell megoldani, akkor alkalmazzuk az LU dekompozíciót, mert ennél a módszernél a jobb oldal nem transzformálódik, így akárhány jobb oldallal is megoldhatjuk az egyeneletrendszert.

Az **A** mátrixot felbontjuk két háromszög mátrixra **L** (Lower = alsó) és egy **U** (Upper = felső) mátrixra:

$$\mathbf{A \; x = B}$$

Behelyettesítve az **A** mátrix helyére az **L U** mátrixszorzatot:

$$\mathbf{L \; U \; x = B}$$

Ahol az **L** mátrix (elemeit r-rel jelöljük):

$$\begin{bmatrix} 1 & 0 & . & 0 & 0 \\ r_{2,1} & 1 & . & 0 & 0 \\ . & . & 1 & 0 & 0 \\ . & . & . & 1 & 0 \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & 1 \end{bmatrix}$$

Ahol az **U** mátrix

$$\begin{bmatrix} u_{1,1} & u_{1,2} & . & . & u_{1,n} \\ 0 & u_{2,2} & . & . & u_{2,n} \\ . & . & . & . & . \\ 0 & 0 & 0 & u_{n-1,n-1} & u_{n-1,n} \\ 0 & 0 & 0 & 0 & u_{n,n} \end{bmatrix}$$

Az n-ed rendű **LU** mátrix n-1 iteráció eredményeként jön létre. Az iteráció folyamán először a főátlóbeli elem alatti elemeket osztjuk a főátlóbeli elemmel, majd a mátrix további elemeit módosítjuk a módosított elemekkel az alábbi módon:

$$a_{i,k}^{(k)} = \frac{a_{i,k}^{(k-1)}}{a_{k,k}^{(k-1)}} \qquad k = 1,2, \ldots n-1$$

$$a_{i,j}^{(k)} = a_{i,j}^{(k-1)} - a_{i,k}^{(k)} \; a_{k,j}^{(k-1)} \qquad \begin{matrix} i = k+1, \ldots n \\ j = k+1, \ldots n \end{matrix}$$

Az iteráció eredményeként a két mátrix (**L** és **U**) együttesen kerül tárolásra, de az 1 (egy értékű) főátlóbeli elemet figyelembe véve:

$$\begin{bmatrix} u_{1,1} & u_{1,2} & . & . & u_{1,n} \\ r_{2,1} & u_{2,2} & . & . & u_{2,n} \\ . & . & . & . & . \\ r_{n-1,1} & . & . & u_{n-1,n-1} & u_{n,n-1} \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & u_{n,n} \end{bmatrix}$$

Az LU dekompozíció bemutatása általánosan

Az **A** 4x4-es mátrix LU dekompozícióját 3 (n-1) iterációs lépés után kapjuk meg. Általánosan a feladat megoldása a következő:

$$\mathbf{A} = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix}$$

Az **A** mátrix elemeinek módosulása az első iterációs lépés után (k = 1):

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a_{2,2}-\frac{a_{2,1}}{a_{1,1}}a_{1,2}=a'_{2,2} & a_{2,3}-\frac{a_{2,1}}{a_{1,1}}a_{1,3}=a'_{2,3} & a_{2,4}-\frac{a_{2,1}}{a_{1,1}}a_{1,4}=a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & a_{3,2}-\frac{a_{3,1}}{a_{1,1}}a_{1,2}=a'_{3,2} & a_{3,3}-\frac{a_{3,1}}{a_{1,1}}a_{1,3}=a'_{3,3} & a_{3,4}-\frac{a_{3,1}}{a_{1,1}}a_{1,4}=a'_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & a_{4,2}-\frac{a_{4,1}}{a_{1,1}}a_{1,2}=a'_{4,2} & a_{4,3}-\frac{a_{4,1}}{a_{1,1}}a_{1,3}=a'_{4,3} & a_{4,4}-\frac{a_{4,1}}{a_{1,1}}a_{1,4}=a'_{4,4} \end{bmatrix}$$

Az **A** mátrix elemeinek módosulása a második iterációs lépés után (k = 2):

$$\begin{bmatrix} a_{1,1} & a_{2,1} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & \frac{a'_{3,2}}{a'_{2,2}}=a''_{3,2} & a'_{3,3}-a''_{3,2}\,a'_{2,3}=a''_{3,3} & a'_{3,4}-a''_{3,2}\,a'_{2,4}=a''_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & \frac{a'_{4,2}}{a'_{2,2}}=a''_{4,2} & a'_{4,3}-a''_{4,2}\,a'_{2,3}=a''_{4,3} & a'_{4,4}-a''_{4,2}\,a'_{2,4}=a''_{4,4} \end{bmatrix}$$

Az **A** mátrix elemeinek módosulása a harmadik iterációs lépés után (k = 3):

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ \frac{a_{2,1}}{a_{1,1}} & a'_{2,2} & a'_{2,3} & a'_{2,4} \\ \frac{a_{3,1}}{a_{1,1}} & \frac{a'_{3,2}}{a'_{2,2}} & a''_{3,3} & a''_{3,4} \\ \frac{a_{4,1}}{a_{1,1}} & \frac{a'_{4,2}}{a'_{2,2}} & \frac{a''_{4,3}}{a''_{3,3}}=a'''_{4,3} & a''_{4,4}-a'''_{4,3}\,a''_{3,4}=a'''_{4,4} \end{bmatrix}$$

$$\mathbf{A\ x = L\ U\ x = L\ \underbrace{(U\ x)}_{\mathbf{y}} = \tilde{L}\ y = b}$$

Először oldjuk meg az

L y = b

lineáris egyenletrendszert

$$\begin{bmatrix} 1 & 0 & 0 & . & 0 \\ r_{2,1} & 1 & . & . & 0 \\ . & . & 1 & 0 & 0 \\ . & . & . & 1 & 0 \\ r_{n,1} & r_{n,2} & . & r_{n,n-1} & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix}$$

Az egyenletből határozzuk meg az **y** vektort. Az első egyenletből kiindulva

$$y_1 = b_1$$

a további ismeretlenek meghatározása:

$$y_k = b_k - \sum_{j=1}^{k-1} r_{k,j}\ y_j, \qquad k = 2, 3, \ldots n$$

Majd oldjuk meg az

U x = y

lineáris egyenletrendszert. Az **y** vektor ismeretében az **x** vektor meghatározható.

$$\begin{bmatrix} u_{1,1} & u_{1,2} & . . & u_{1,n} \\ 0 & u_{2,2} & . . & u_{2,n} \\ 0 & 0 & . . & . \\ 0 & . & . & u_{n-1,n-1} \quad u_{n-1,n} \\ 0 & . & . & 0 \quad u_{n,n} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ x_n \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ . \\ . \\ y_n \end{bmatrix}$$

Az utolsó egyenletből kiindulva:

$$x_n = \frac{y_n}{u_{n,n}}$$

a további ismeretlenek meghatározása:

$$x_k = \frac{y_k - \sum_{j=k+1}^{n} u_{k,j} \; x_j}{u_{k,k}}, \qquad k = n-1, n-2, \ldots 1$$

Az LU.C program függvényei:

```
/* A mátrix sorainak felcserélése */
void sorcsere(tomb a,tomb e,int xm,int k,int n)
{
  int i;
  double c;
  for ( i = 1; i <= n+1 ; i++ )
  {
    c=a[xm][i]; a[xm][i]=a[k][i]; a[k][i]=c;
    c=e[xm][i]; e[xm][i]=e[k][i]; e[k][i]=c;
  }
}

/* A főelem kiválasztása */
void foelem(tomb a,tomb e,int n,int k, int *d)
{
  int i,xm;
  double am;
  am=a[k][k]; xm=k;
  for ( i = k; i <= n; i++ )
  {
    if (fabs(am) < fabs(a[i][k]))
    {
      am=a[i][k]; xm=i;
      sorcsere(a,e,xm,k,n);
      *d=*d *(-1);
    }
  }
}
```



```
/* dekompozició végrehajtása */
void dekomp(tomb a,int n, int *d)
{
  int i,j,k;

   *d = 1;
   for ( k = 1; k < n; k++ )
   {
      printf("\n%d. iterácio Oszt: %10.3lf\n\n",k,a[k][k]);
      for ( i = k+1; i <= n; i++ )
      {
      a[i][k]=a[i][k]/a[k][k];
      for ( j = k+1; j <= n; j++ )
      {
        a[i][j]=a[i][j]-a[i][k]*a[k][j];
      }
      }
      matrixkiir(a,n);
   }
}


/* dekompozició végrehajtása főelem kiválasztásával */
void dekompf(tomb a,tomb e,int n, int *d)
{
 int i,j,k;

   *d = 1;
   for ( k = 1; k < n; k++ )
   {
      foelem(a,e,n,k,d);
      printf("\n%d. iterácio Oszt: %10.3lf\n\n",k,a[k][k]);
      for ( i = k+1; i <= n; i++ )
      {
        a[i][k]=a[i][k]/a[k][k];
        for ( j = k+1; j <= n; j++ )
        {
          a[i][j]=a[i][j]-a[i][k]*a[k][j];
        }
      }
      matrixkiir(a,n);
   }
}
```

```
/*  Ly = b egyenlet megoldása */
void yvektor(vekt y,tömb a,int n)
{
  int j,k;
  y[1]=a[1][n+1];
  for ( k = 2; k <= n; k++)
  {
    y[k]=0;
    for ( j = 1; j <= k-1; j++)
    {
      y[k]=y[k]-a[k][j]*y[j];
    }
    y[k]=y[k]+a[k][n+1];
  }
}

/*  U x = y egyenlet megoldása */
void xvektor(vekt x,vekt y,tomb a,int n)
{
  int j,k;
    x[n]=y[n]/a[n][n];
    for ( k = n-1; k >= 1; k--)
    {
      x[k]=0;
      for ( j = k+1; j <= n; j++ )
      {
        x[k]=x[k]-a[k][j]*x[j];
      }
      x[k]=(y[k]+x[k])/a[k][k];
    }
}
```

4. Feladat: Oldjuk meg az 1. feladatban megadott egyenletrendszert LU dekompozícióval. Az LU.C program eredménye:

```
1.  Fõelemkiválasztás nélkül
2   Részleges fõelemkiválasztással
3.  Kilépés

Menü kiválasztása: 1

Elemek száma [max. 20]: 3

a[1,1]= 10
a[1,2]= 5
a[1,3]= 2

a[2,1]= 20
a[2,2]= 12
a[2,3]= 7

a[3,1]= 30
a[3,2]= 19
a[3,3]= 17
```



```
b[1]= 9
b[2]= 22
b[3]= 45

A mátrix

    10.000     5.000     2.000
    20.000    12.000     7.000
    30.000    19.000    17.000

1. iterácio Oszt:     10.000

    10.000     5.000     2.000
     2.000     2.000     3.000
     3.000     4.000    11.000

2. iterácio Oszt:      2.000

    10.000     5.000     2.000
     2.000     2.000     3.000
     3.000     2.000     5.000

Az egyenlet gyökei

x[1]=  1.0000
x[2]=  -1.0000
x[3]=  2.0000

Az inverz mátrix:

     0.710    -0.470     0.110
    -1.300     1.100    -0.300
     0.200    -0.400     0.200

A determináns: 100.000000
```

5. Feladat: Futtassuk le az LU.C programot részleges főelemkiválasztással:

```
A x = b lineáris egyenletrendszer megoldása
          LU dekompozicióval

1.  Fõelemkiválasztás nélkül
2.  Részleges fõelemkiválasztással
3.  Kilépés

Menü kiválasztása: 2
Elemek száma [max. 20]: 3

a[1,1]= 10
a[1,2]= 5
a[1,3]= 2
```

```
a[2,1]= 20
a[2,2]= 12
a[2,3]= 7

a[3,1]= 30
a[3,2]= 19
a[3,3]= 17

b[1]= 9
b[2]= 22
b[3]= 45

A mátrix

    10.000     5.000     2.000
    20.000    12.000     7.000
    30.000    19.000    17.000

1. iterácio Oszt:     30.000

    30.000    19.000    17.000
     0.333    -1.333    -3.667
     0.667    -0.667    -4.333

2. iterácio Oszt:     -1.333

    30.000    19.000    17.000
     0.333    -1.333    -3.667
     0.667     0.500    -2.500

Az egyenlet gyökei

x[1]=  1.0000
x[2]=  -1.0000
x[3]=  2.0000

Az inverz mátrix:

     0.710    -0.470     0.110
    -1.300     1.100    -0.300
     0.200    -0.400     0.200
```

5.1.6. Mátrixinvertálás

Ha az **A** mátrixnak **B** mátrix az inverze, akkor a

$$\mathbf{B}\,\mathbf{A} = \mathbf{E}$$

szorzat egységmátrixot kell, hogy adjon.

Ahol az **E** egységmátrix

$$E = \begin{bmatrix} 1 & 0 & . & . & 0 \\ 0 & 1 & . & . & 0 \\ . & . & 1 & . & 0 \\ 0 & 0 & . & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Mátrixinvertálás LU dekompozícióval

A mátrix inverzének meghatározásához használjuk fel az LU dekompozíció azon tulajdonságát, hogy az egyszer dekomponált mátrixot *n*-szer megoldjuk az **E** egységmátrix oszlopvektoraival.

$$\mathbf{L}\,\mathbf{U}\,\mathbf{x} = e_i \quad \text{ahol} \quad i = 1,2, \ldots n$$

$$e_1 = \begin{bmatrix} 1 \\ 0 \\ . \\ . \\ 0 \end{bmatrix} \quad e_2 = \begin{bmatrix} 0 \\ 1 \\ . \\ . \\ 0 \end{bmatrix} \quad \ldots \quad e_n = \begin{bmatrix} 0 \\ 0 \\ . \\ . \\ 1 \end{bmatrix}$$

A kapott gyökök vektorai szolgáltatják az invertált mátrix oszlopvektorait. A sorozatosan kapott eredményt betöltjük a **B** mátrix oszlopaiba, és az *n*-edk megoldás után előáll a **B** mátrixban az **A** mátrix inverze.

```
/* A mátrix inverzének kiszámitása */
void inverz(tomb a,tomb e, int n)
{
 int i,j;
 vekt x, y;
  for ( j = 1; j <= n; j++ )
  {
    for ( i = 1; i <= n; i++ )
    {
      a[i][n+1]=e[i][j];
    }
    yvektor(y,a,n);
```

```
    xvektor(x,y,a,n);
    for ( i = 1; i <= n; i++ )
    {
      e[i][j]=x[i];
    }
  }
  matrixkiir(e,n);
}

/* A determináns kiszámitása */
double det(tomb a,int n, int d)
{
  int k;
  double c;
  c=a[1][1];
  for ( k = 2; k <= n; k++ )
  {
    c=c*a[k][k];
  }
  return (d * c);
}
```

5. Feladat: Invertáljuk LU dekompozícióval az alábbi mátrixot:

$$\begin{bmatrix} 0,0001x_1 & x_2 & x_3 \\ x_1 & 0,00001x_2 & x_3 \\ x_1 & 4x_2 & 0,00001x_3 \end{bmatrix}$$

Futtasuk le az INVERZ.C programot. A program eredménye:

```
Mátrix invertálása LU dekompozicióval
   részleges fõelemkiválasztással


Elemek száma [max. 20]: 3

a[1,1]= 0.0001
a[1,2]= 1
a[1,3]= 1

a[2,1]= 1
a[2,2]= 0.00001
a[2,3]= 1

a[3,1]= 1
a[3,2]= 4
a[3,3]= 0.00001
```



```
A mátrix

   1 0000e-04   1.0000e+00   1.0000e+00
   1.0000e+00   1.0000e-05   1.0000e+00
   1.0000e+00   4.0000e+00   1.0000e-05

1. iterácio Oszt:    1.0000e+00

   1.0000e+00   1.0000e-05   1.0000e+00
   1.0000e-04   1.0000e+00   9.9990e-01
   1.0000e+00   4.0000e+00  -9.9999e-01

2. iterácio Oszt:    4.0000e+00

   1.0000e+00   1.0000e-05   1.0000e+00
   1.0000e+00   4.0000e+00  -9.9999e-01
   1.0000e-04   2.5000e-01   1.2499e+00

Az inverz mátrix:

  -8.0007e-01   8.0007e-01   2.0001e-01
   2.0001e-01  -2.0002e-01   2.0000e-01
   8.0007e-01   1.9994e-01  -2.0002e-01

A determináns: 4.999580e+00
```

5.2. Egyismeretlenes nemlineáris egyenlet megoldása

A feladat az $f(x) = 0$ tetszőleges, nemlineáris egyenlet gyökeinek meghatározása.
A gyakorlatban mindig teszünk kikötéseket f(x)-re és a keresett gyök (gyökök) elhelyezkedésére vonatkozóan. A következőkben valós gyökök meghatározásával foglalkozunk.

A gyökök elkülönítése

Vannak olyan módszerek, amelyek feltételezik, hogy a vizsgált intervallumban f(x)-nek csak egyetlen gyöke van. Ilyen módszer alkalmazásakor az [a,b] intervallumban keresünk olyan részintervallumokat, amelyre igaz, hogy bennük csak egy-egy gyök van. Ez a gyökök elkülönítése. Ezt gyakran összekapcsoljuk annak vizsgálatával, hogy egyáltalán van-e gyök.

Grafikus

A gyökök elkülönítését végezhetjük grafikusan is. Az egyenlet valós gyökeinek számáról és elhelyezkedéséről tájékozódhatunk, ha az f(x) függvény görbéjét felvázoljuk. Az x tengellyel való metszéspontok és érintési pontok szolgáltatják az egyenlet gyökeit.

Gyakran az $y = f(x)$ függvény nehezen ábrázolható, de f(x) felbontható

$$f(x) = u(x) - v(x)$$

alakban, és

$$y_1 = u(x)$$

valamint

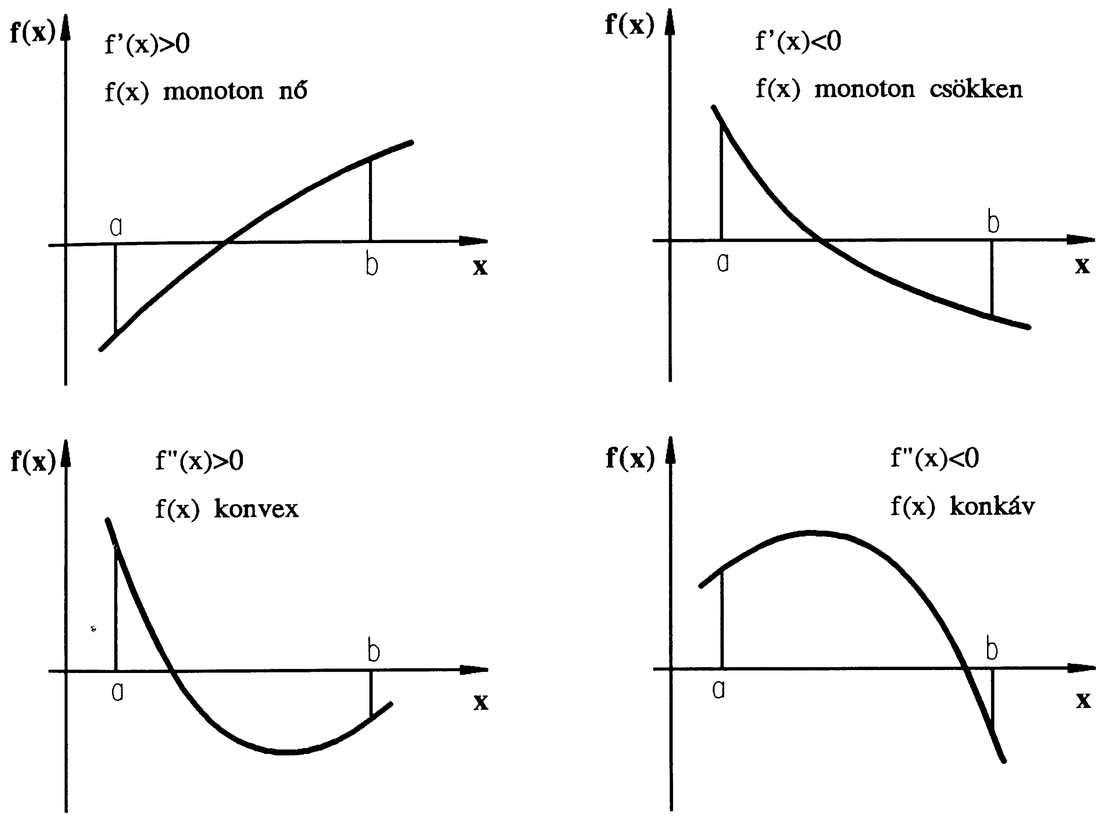
$$y_2 = v(x)$$

egyszerűbben ábrázolható. Ilyenkor a gyökök a görbék metszéspontjai.

Bolzano tétel

A gyökök elkülönítéséhez segítséget nyújt Bolzano tétele. Ha egy f(x) függvény az [a,b] intervallumban folytonos, valamint f(a) és f(b) ellenkező előjelűek (vagyis $f(a)\ f(b) < 0$), akkor f(x)-nek létezik legalább egy gyöke az [a,b] intervallumban.

Ha Bolzano tétele teljesül és az $f'(x) < 0$, $f'(x) > 0$, $f''(x) < 0$, illetve $f''(x)>0$ az egész [a,b] intervallumban feltételek közül legalább egy szintén teljesül, akkor az [a,b] intervallumban csak egy gyök van.
Szemléletesen (bizonyítás nélkül):



5 1 ábra Bolzano tétel szemléltetése

Numerikus módszer

Numerikusan úgy járunk el, hogy addig keresünk részintervallumokat, amíg be nem bizonyítjuk, hogy csak egy-egy gyök van bennük, illetve egyetlen gyököt sem tartalmaznak.

Tekintsünk egy hasznos numerikus eljárást. Számítsuk ki az

$$x_i = a + i \frac{b - a}{k} \qquad i = 0,1,2, \ldots k$$

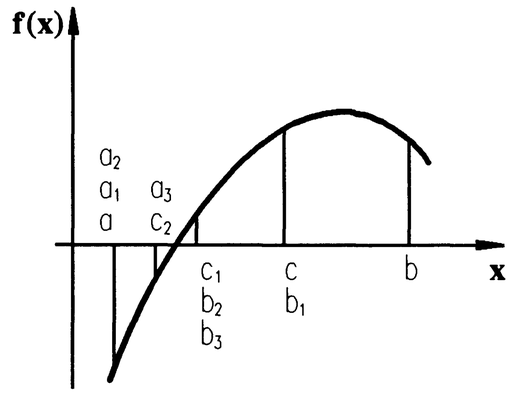
pontokban f(x) értékét (k>2, például 10) és vizsgáljuk meg, hogy $f(x_i)f(x_{i+1}) < 0$ teljesül-e. Ha igen, akkor k értékét növeljük (pl. 10-szeresére). Ezzel eldönthető, hogy van-e gyök és körülbelül hol helyezkedik el.

Az egyismeretlenes nemlineáris egyenletek megoldására több módszer közül választhatunk:

- behatárolás intervallum-felezéssel,
- érintő módszer (Newton-Raphson),
- húr módszer,
- Newton-Raphson és a húr módszer együttes alkalmazása,
- szelő módszer,
- fokozatos közelítés (szukcesszív approximáció).

5.2.1. Gyök behatárolása intervallum-felezéssel

Tételezzük fel, hogy az f(x) függvény az [a,b] intervallumban folytonos és az intervallum két végpontján a függvényértékek ellentétes előjelűek (f(a)f(b)<0). Ekkor bebizonyítható, hogy egy (vagy páratlan számú) zérushellyel rendelkezik az f(x) függvény az [a,b] intervallumban. Közvetlenül ezt a tényt használja ki az intervallum felezéses eljárás.



5 2 ábra Gyök behatárolása intervallum felezéssel

Az eljárás a következő:

- Felezzük meg az [a,b] intervallumot

$$c = \frac{a + b}{2}$$

- Nézzük meg, hogy f(c) zérus-e. Ha zérus, akkor c a gyök.

- Ha nem zérus, akkor nézzük meg, hogy az f(c) előjele f(a) előjelétől vagy f(b) előjelétől különbözik-e:

  ha $f(a)\ f(c) < 0$ akkor

  $$a_1 = a \qquad b_1 = c$$

  ha viszont $f(b) \cdot f(c) > 0$, akkor a

  $$a_1 = c \qquad b_1 = b$$

  választást alkalmazzuk.

- ha $|a_1 - b_1| > \varepsilon$, akkor folytatjuk az eljárást az új a_1 és b_1 intervallumban. Amennyiben $|a_1 - b_1| < \varepsilon$, akkor az

  $$\frac{a_1 + b_1}{2}$$

  értéket gyöknek tekintjük.

A módszer konvergenciájára jellemző, hogy

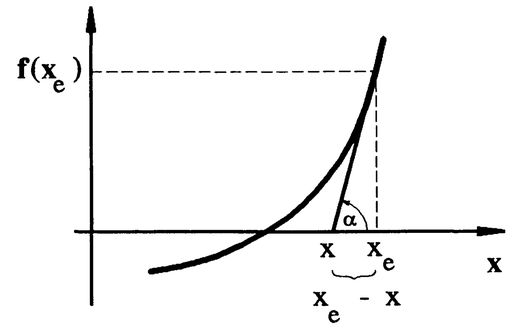
$$b_n - a_n = \left(\frac{1}{2^n}\right)(b - a)$$

Mivel a módszer konvergenciája lassú, ezért a gyök közelítő értékének behatárolására használjuk, majd ezután a gyök pontos értékének meghatározására például a Newton - Raphson módszert alkalmazzuk.
A módszert használó programrészletet alkalmazzuk:

```
int Felezo(double a,      /* intervallum eleje      */
        double b,         /* intervallum vége       */
        double eps,       /* hibakorlát             */
        double *gyok,     /* gyök -> eredmény       */
        dfptr f)          /* az f(x) függvény cime */
{
    double fa, fb, x, fx;

    fa = (*f)(a); fb = (*f)(b);
    *gyok = 0.0;

    if (fa * fb > 0.0) return 0;
    fx = fabs(fa);
    while (fx > eps)
    {
      x = (a+b)/2;
      fx = (*f)(x);
      if (fx * fa > 0.0) { a = x; fa = fx; } else b = x;
      fx = fabs(fx);
    }
    *gyok = x;
    return 1;
}
```

5.2.2. Gyök meghatározása érintő módszerrel (Newton-Raphson módszer)

Az érintő módszer (Newton-Raphson módszer) az [a,b] intervallumban folytonos függvényt érintőjével helyettesíti és az érintő x tengellyel való metszéspontját tekinti a gyökhely közelítésének. E módszer sorozatos alkalmazása - bizonyos feltételek teljesülése esetén - gyorsan, négyzetesen konvergál. A Newton-Raphson módszer alkalmazásához az f(x) függvényre és annak x szerinti differenciálhányadosára is szükség van.



5 3 ábra Gyök meghatározása érintő módszerrel

Fejezzük ki az érintő meredekségét kétféle módon:

$$\mathrm{tg}\alpha = f'(x_e)$$

$$\mathrm{tg}\alpha = \frac{f(x_e)}{x_e - x}$$

A két egyenletet egyenlővé téve

$$\frac{f(x_e)}{x_e - x} = f'(x_e) \ .$$

Az egyenletet az x közelítő gyökre rendezve

$$x = x_e - \frac{f(x_e)}{f'(x_e)}$$

A leállási feltétel

$$|x - x_e| < \varepsilon \ .$$

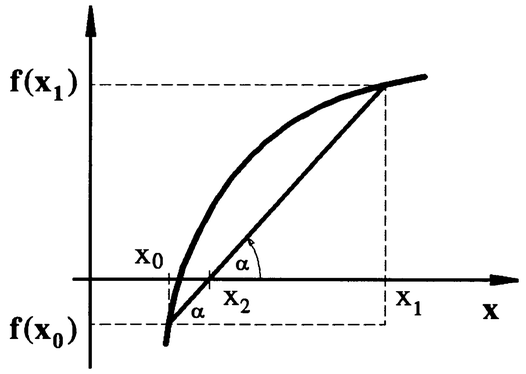
Négyzetes a konvergencia az egyszeres gyökökre.
Bizonyítás nélkül: az [a,b] intervallum két végpontja közül azt tekintjük kiindulási pontnak, melyben $f(x)\ f''(x) > 0$.

A módszert használó programrészletet alkalmazzuk:

```
int NewtonR(double x0,  /* intervallum eleje       */
        double eps,     /* hibakorlát              */
        double *gyok,   /* gyök -> eredmény        */
        dfptr  f,       /* az f(x)  függvény cime  */
        dfptr  df)      /* az f'(x) függvény cime  */
{
    double xe, nevezo;

    xe = x0;
    do
    {
     nevezo = (*df)(xe);
     if (fabs(nevezo) < 1e-6) return 0;
     xe = xe - (*f)(xe)/nevezo;
    }
    while(eps < fabs((*f)(xe)));

    *gyok = xe;
    return 1;
}
```

5.2.3 Húr módszer

A Newton-Raphson módszer alkalmazását megnehezíti, hogy a függvény differenciálhányadosának kiszámítására is szükség van. Előnyösebbek azok az algoritmusok, melyeknek konvergenciája kellőképpen gyors, azonban számításigényük kevesebb. A legjellegzetesebb ilyen interpolációs módszer a húr módszer, amely szintén két alappontra támaszkodik, és az alappontok között a függvényt egyenessel helyettesítik



5 4 ábra Gyök meghatározása húr módszerrel

Szükséges feltétel, hogy az intervallum két végpontjában a függvényértékek ellenkező előjelűek legyenek:

$$f(x_0)\ f(x_1) < 0$$

Írjuk fel a húr meredekségét kétfajta módon:

$$\mathrm{tg}\alpha = \frac{f(x_1) - f(x_0)}{x_1 - x_0}$$

$$\mathrm{tg}\alpha = \frac{f(x_1)}{x_1 - x_2}$$

Egyenlővé téve a két egyenlőséget, az alábbi egyenletet kapjuk:

$$\frac{f(x_1) - f(x_0)}{x_1 - x_0} = \frac{f(x_1)}{x_1 - x_2}$$

Legyen

$$y_0 = f(x_0),\quad y_1 = f(x_1)$$

akkor x_2 közelítő gyök értéke

$$x_2 = x_1 - y_1 \frac{x_1 - x_0}{y_1 - y_0}$$

a leállási feltétel pedig az

$$|x_2 - x_1| \le \varepsilon$$

Az intervallum módosítása az y_2 előjele szerint történik, ha az y_2 előjele y_1 előjelével azonos, akkor

$$x_1 = x_2,\ y_1 = y_2$$

különben

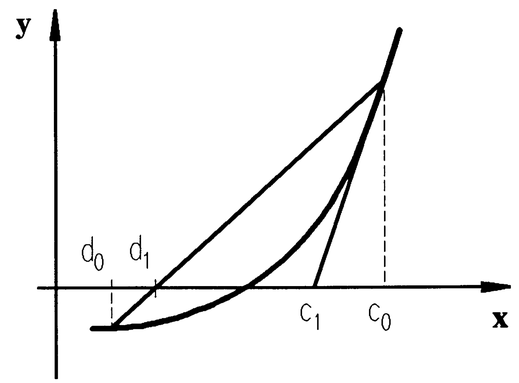
$$x_0 = x_2,\quad y_0 = y_2$$

A módszert használó prøgramrészletet alkalmazzuk:

```
int Hur(double a,double b,      /* intervallum eleje és vége      */
        double eps,             /* hibakorlát                     */
        double *gyok,           /* gyök -> eredmény               */
        dfptr  f)               /* az f(x) függvény cime          */
{
 double x0, x1, x2, y0, y1;
   *gyok = 0.0;
   x0 = a;
   x1 = b;
   if(((*f)(x1)*(*f)(x0)) < 0)
   {
     do
     {
     y0 = (*f)(x0);
     y1 = (*f)(x1);
     x2 = x1 - y1*(x1-x0)/(y1-y0);
     if ((*f)(x2)*y1 > 0) x1 = x2;
     else x0 = x2;
     }
     while(eps < fabs((*f)(x2)));
     *gyok = x2;
     return 1;
  } else return 0;
}
```

5.2.4. Gyök meghatározása a Newton-Raphson módszer és a húr módszer együttes alkalmazásával

A Newton-Raphson és a húr módszert egyszerre alkalmazva, gyorsabb konvergenciát kapunk. Ugyancsak teljesülni kell annak a feltételnek, hogy a függvény az intervallum két oldalán ellenkező előjellel rendelkezik.



5 5 ábra Gyök meghatározása Newton-Raphson és a húr módszer együttes alkalmazásával

Newton-Raphson módszerrel a közelítő gyök

$$c_{k+1} = c_k - \frac{f(c_k)}{f'(c_k)}$$

Húr módszerrel a közelítő gyök

$$d_{k+1} = d_k - f(d_k)\frac{d_k - c_k}{f(d_k) - f(c_k)}$$

A leállási feltétel:

$$|c_{k+1} - d_{k+1}| < \varepsilon$$

A módszert használó programrészletet alkalmazzuk:

```
int NewtonRH(double c,double d,     /* intervallum eleje és vége  */
             double eps,            /* hibakorlát                 */
             double *gyok,          /* gyök -> eredmény           */
             dfptr  f,              /* az f(x)  függvény cime     */
             dfptr  df)             /* az f'(x) függvény cime     */
{
   double ce, de, fce, fde, fdce;
   *gyok = 0.0;
```

```
   if (((*f)(d)*(*f)(c)) < 0.0)
   {
     ce = c;
     de = d;
     do
     {
       fce = (*f)(ce);
       fde = (*f)(de);
       fdce = (*df)(ce);
       if (fabs(fde-fce) < 1e-6) return 0;
       if (fabs(fdce) < 1e-6) return 0;
       de = de - fde*(de-ce)/(fde-fce);
       ce = ce -fce/fdce;
     }
     while(eps < fabs(de-ce));
     *gyok = de;
     return 1;
  } else return 0;
}
```

5.2.5. *Gyök meghatározása szelő módszerrel*

Gyakran előfordul, hogy a Newton-Raphson képletben szereplő deriváltakat nehéz lenne meghatározni - viszont gyors konvergenciára törekszünk. Ilyenkor segít a következő iterációs módszer, amely a szelő módszer vagy regula-falsi módszer nevet viseli:

$$x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$$

A módszert használó programrészletet alkalmazzuk:

```
int Szelo(double a, double b,  /* intervallum eleje és vége  */
       double eps,             /* hibakorlát                 */
       double *gyok,           /* gyök -> eredmény           */
       dfptr  f)               /* az f(x) függvény cime      */
{
   double xe, xe1, xe2;
   *gyok = 0.0;

   if(((*f)(a)*(*f)(b)) < 0)
   {
     xe = a;
     xe1 = b;
     do
     {
```



```
       xe2 = xe1 - (*f)(xe1)*(xe1-xe)/((*f)(xe1)-(*f)(xe));
       xe = xe1;
       xe1 = xe2;
     }
     while(eps < fabs((*f)(xe2)));
     *gyok = xe2;
     return 1;
  } else return 0;
}
```

5.2.6. *Gyök meghatározása fokozatos közelítéssel*

A fokozatos közelítés (szukcesszív approximáció) vagy más szóval az iterálás módszere nemcsak egyenletek megoldásának egyik legfontosabb eljárása. A

$0 = f(x)$ egyenlet megoldását keressük.

Írjuk át az egyenletet a következő alakra:

$x = g(x)$, ahol $g(x) = f(x) + x$

Ha $f(x_p) = 0$ (vagyis x_p gyök), akkor

$x_p = g(x_p)$

Ha x_0 nem gyöke az egyenletnek, akkor x_1-t kapunk

$x_1 = g(x_0)$

$x_2 = g(x_1)$

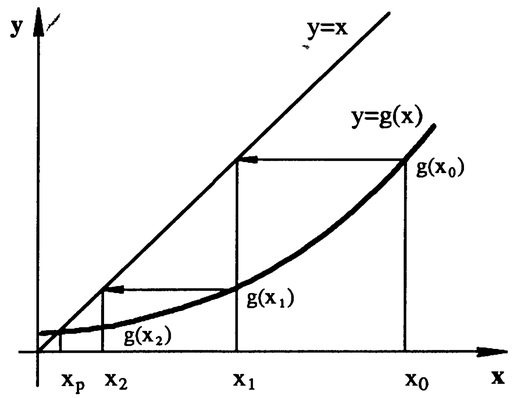
. . ,

általánosságban egy sorozat keletkezik:

$x_{i+1} = g(x_i)$ ($i = 1, 2, ...$)

Ha ez a sorozat konvergens, akkor a sorozat határértéke a keresett gyök:

$$\lim_{i\to\infty} x_i = x_p$$



5 6 ábra Gyök meghatározása fokozatos közelítéssel

A konvergencia feltétele (bizonyítás nélkül):

- g(x) az [a,b] intervallumban differenciálható és

- $\left|g'(x)\right| \leq q < 1$ az (a,b) intervallumban

Ebben az esetben a konvergencia nem függ attól, hogy az x_0 kezdőérték az [a,b] intervallumban hová esik, de benne kell lennie.

6. Feladat: Határozzuk meg az

$$y = x^3 + x - 0,7$$

nemlineáris egyenlet a gyökét a [0,1] intervallumban 1e-6 hibakorláttal.
A NEMLIN.C programban módosítsuk az *fg* és a *dfg* függvényeket:

```
double fg(double x)
{
   return (x*x*x+x-0.7);
}

double dfg(double x)
{
  return (3*x*x+1);
}
```


A NEMLIN.C program futtatási eredménye:

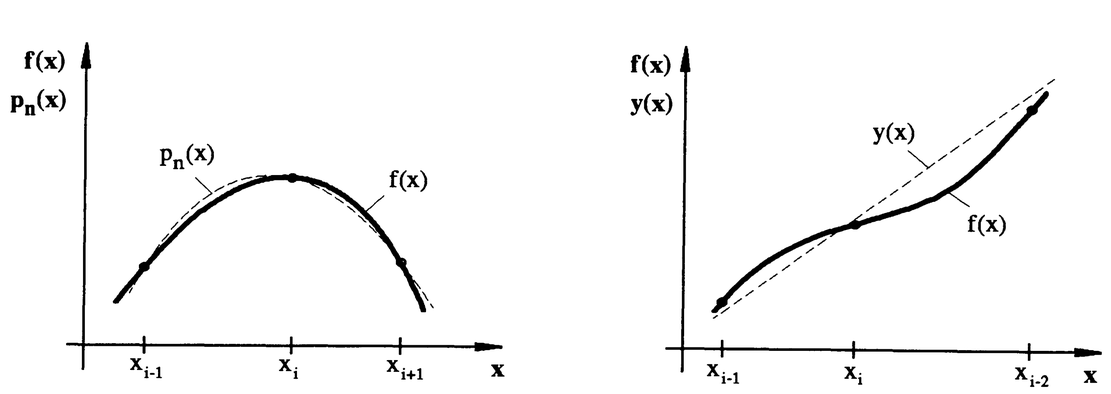
```
Nemlineáris egyenlet megoldása

Intervallum eleje : 0
Intervallum vége  : 1
Hibakorlát        : 1e-6

Módszerek:

1: Intervallum felezés
2: Hur módszer
3: Newton-Raphson módszer
4: Newton-Raphson és a Húr módszer együtt
5: Szelõ módszer

Módszer száma : 3

Newton_Raphson módszer
A nemlineáris egyenlet gyöke : 0.541351
```

5.3. Interpoláció, regresszió

Az interpoláció és a regresszió függvények közelítését jelenti függvényekkel, általában polinomokkal. Az alábbi esetekben használjuk:

- A racionális függvények értékeit közvetlenül tudjuk számítani, de az esetleg hosszadalmas bonyolult számítás helyett egy egyszerűbb függvénnyel célszerűbb közelíteni.
- Az irracionális függvényeket racionálisokkal, főleg polinomokkal közelítjük és ezek értékeit számítjuk.
- Mérési sorozatok esetén a függvény értékeit véges számú pontban ismerjük, a mérési pontok között a függvényt közelítjük.
- A függvények és a mérési eredmények ábrázolásánál véges számú pontot veszünk fel és ezeket kötjük össze.

Ha a közelítő függvény értéke az ismert alappontokban megegyezik a közelített függvény értékével, akkor interpolációról beszélünk. A regresszióról van szó, ha a közelítő függvény az alappontokban nem szükségszerűen egyezik meg a közelítettel, a cél csak a vizsgált tartományon belül egy minimális hiba biztosítása.



Interpoláció *Regresszió*

5 7 ábra A függvények közelítési módszerei

5.3.1. Interpoláció

A függvények grafikus ábrázolása olyan terület, ahol lépten-nyomon találkozunk az interpoláció szükségességével. Ha néhány pontjával megadott függvényt kell folytonos görbével ábrázolnunk, a függvény ismeretlen pontjait mindig valamilyen interpolációs módszerrel határozzuk meg.
Tegyük fel, hogy az $f(x)$ függvény értékét csak az

$$x = x_0, x_1, \ldots, x_n$$

pontokban, az úgynevezett interpolációs alappontokban ismerjük. Az $f(x)$ függvényt az $[x_0, x_n]$ intervallumban a $p_n(x)$ interpolációs függvénnyel (p_n-polinommal) helyettesítjük. A $p_n(x)$ polinom előállításánál az a feladatunk, hogy a közelítő polinom az alappontokban vegye fel a függvényértékeket, az alappontokon kívül pedig ε-nál jobban közelítsen, vagyis

$$p_n(x_i) = f(x_i)$$
$$\left|p_n(x) - f(x)\right| \leq \varepsilon$$

Az interpoláció egyik alapkérdése, hogy egy például $[x_{i-1}, x_i]$ tartományon belül a függvény közelítéséhez hány alappontot használjunk fel, (például x_{i-2}, x_{i-1}, x_i, x_{i+1} esetén 4 alappontot), és ezek milyen súllyal vegyenek részt a közelítő függvény elkészítésében. Különösebb bizonyítás nélkül is belátható, hogy az egész tartomány n alappontjára illesztett n-edfokú polinom rendkívül bonyolult, ezzel visszajutunk az eredeti problémához (t.i. a bonyolultsághoz). Ugyanakkor nem biztos, hogy ténylegesen jó eredményt szolgáltat. A probléma úgy oldható meg, hogy az egész tartományt több résztartományra bontjuk fel, és ezeken belül kisebb fokszámú közelítéseket használunk.

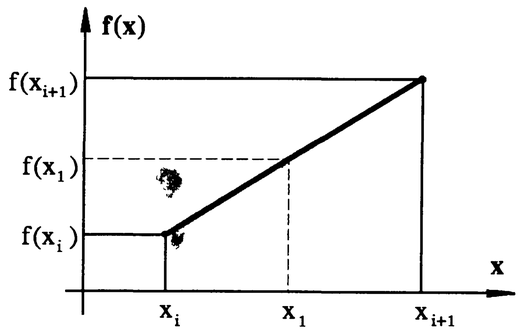
Az interpoláció másik alapkérdése, hogy az interpolációs alappontokat hova vegyük fel (amennyiben lehetőség van a választásra) ahhoz, hogy a legkisebb fokszámú közelítéssel a legjobb eredményt érjük el. Bebizonyítható, hogy létezik az alappontok felvételének optimális értéke (Csebisev alappontok) a gyakorlatban egyszerűsége miatt mégis az egyenletes (ekvidisztáns) alappont-elhelyezést:

$$x_i = x_0 + \frac{x_n - x_0}{n} \quad i$$

használják. A későbbiekben mi is ezt az alappont elhelyezést alkalmazzuk.

5.3.2. *Lineáris interpoláció*

A legegyszerűbb (legkisebb fokszámú) közelítés a lineáris interpoláció. Ismert az $(x_i, f(x_i))$, illetve $(x_{i+1}, f(x_{i+1}))$ pontpár, keressük x_1 helyen $f(x_1)$ értéket egyenessel közelítve



5 8 ábra Lineáris interpoláció

$$f(x_1) - f(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \quad (x_1 - x_i)$$

tehát

$$f(x_1) = f(x_i) + \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i} \quad (x_1 - x_i)$$

Lineáris interpolációt végző függvény:

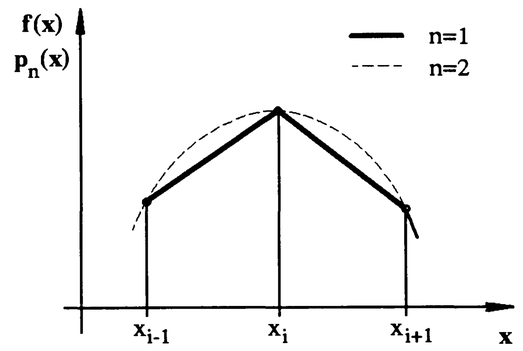
```
int linint(vekt x, vekt y, int n, double x1, double *y1)
{
   int i;

   for(i = 0; i < n-1; i++)
   {
     if((x[i] <= x1) && (x[i+1] >= x1))
     {
      *y1 = y[i]+(y[i+1]-y[i])/(x[i+1]-x[i])*(x1-x[i]);
      return 0;
     }
   }
   return 1;
}
```


A LININTP.C programot futtassuk le az alábbi adatokkal:

```
Lineáris interpoláció

Az alappontok száma: 4
Az alappontok növekvõ sorrendben

x[0] = 1
y[0] = 1

x[1] = 2
y[1] = 1.8

x[2] = 3
y[2] = 2.2

x[3] = 4
y[3] = 2.5

Az interpoláció helye = 2.5
Az interpolált érték  = 2.000000
```

5.3.3. *Lagrange interpoláció*

A két alappontot (x_0, x_1) felhasználó közelítés ($n=1$) a két pont közötti egyenessel közelít. Belátható, hogy a három alappontos közelítés ($n=2$) egy másodfokú függvény, parabola lesz.



5 9 ábra Két alappontot felhasználó közelítés

A négy alappontos közelítés harmadfokú függvény, és így tovább. *n+1* alappont esetén bebizonyítható, hogy a közelítés *n*-edfokú polinom lesz. A polinom együtthatóinak kiszámítására Lagrange dolgozott ki eljárást.

Állítsuk elő a polinomot az alábbi alakban:

$$p_n(x) = \sum_{i=0}^{n} l_i(x) \quad f(x_i)$$

és

$$l_i(x_j) = 0, \quad \text{ha} \quad i \neq j$$

$$l_i(x_j) = 1, \quad \text{ha} \quad i = j$$

vagyis az l_i függvények a saját alappontban 1, a többi alappontban 0 értéket adnak. Akkor

$$p_n(x_i) = 0 \quad f(x_0) + 0 \quad f(x_1) + \ldots + 1 \quad f(x_i) + \ldots + 0 \quad f(x_n)$$

vagyis a feltétel teljesül. A közelítő polinom az alappontban (és valamennyi alappontban) felveszi a közelített függvény értékét, az alappontok között pedig valamilyen közelítést ad. A kérdés már csak az, hogyan lehet az $l_i(x)$ függvényeket előállítani.
Legyen

$$l_i(x) = \frac{(x - x_1) \quad (x - x_2) \ldots (x - x_{i-1}) \quad (x - x_{i+1}) \ldots (x - x_n)}{(x_i - x_1) \quad (x_i - x_2) \ldots \quad (x_i - x_{i-1}) \quad (x_i - x_{i+1}) \ldots (x_i - x_n)}$$

Ekkor az x_1 pontban a számláló első tényezője nulla, x_2 pontban a második és így tovább, tehát mivel csak az $(x - x_i)$ tényező hiányzik, ez minden más alappontban nulla lesz. Az $x = x_i$ helyen pedig a számláló és a nevező azonos, $l_i(x_i) = 1$. Az $l_i(x)$ Lagrange féle interpolációs alappolinomokat tömörebben felírva

$$l_i(x) = \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j} \qquad i = 0, \ldots n$$

alakban írva

$$p_n(x) = \sum_{i=0}^{n} f(x_i) \quad \prod_{\substack{j=0 \\ i \neq j}}^{n} \frac{x - x_j}{x_i - x_j}$$

5.3.3.1. Első fokú Lagrange féle interpolációs polinom

n=1 esetén $l_0(x)$ és $l_1(x)$ létezik

$$l_0(x) = \frac{x - x_1}{x_0 - x_1},$$

$$l_1(x) = \frac{x - x_0}{x_1 - x_0}.$$

Az interpolációs polinom pedig

$$p_1(x) = \frac{x - x_1}{x_0 - x_1} \quad f(x_0) + \frac{x - x_0}{x_1 - x_0} \quad f(x_1)$$

Az egyenlethez 0-t hozzáadva, értéke nem változik,

$$p_1(x) = \frac{x - x_1}{x_0 - x_1} \quad f(x_0) + \frac{x - x_0}{x_1 - x_0} \quad f(x_1) - \frac{x - x_0}{x_1 - x_0} \quad f(x_0)$$

$$+ \frac{x_0 - x}{x_0 - x_1} \quad f(x_0)$$

mivel a negyedik tag számlálója és nevezője is -1-szerese a harmadik tagénak. A második és harmadik tagból a tört kiemelhető, az első és negyedik tagból pedig $f(x_0)$-át, amely $f(x_0)$-ra egyszerűsödik le:

$$p_1(x) = \frac{x - x_0}{x_1 - x_0} \quad (f(x_1) - f(x_0)) + f(x_0)$$

vagyis valóban visszakaptuk a lineáris interpoláció képletét.

A *lagrange* alprogram az *n* alappontban ismert függvény *x1* helyen lévő Lagrange-féle interpolációját adja meg *y1*-ben:

```
int lagrange(double *x, double *y, int n, double x1, double *y1)
{
 double p = 0.0, s;
 int    i, j;

   for(i = 0; i < n; i++)
   {
     s = 1.0;
     for(j = 0; j < n; j++)
     {
       if (i != j)
       {
      if (x[i] == x[j]) return 1;
      else s = s*(x1-x[j])/(x[i]-x[j]);
       }
     }
     p += y[i]*s;
   }
   *y1 = p;
   return 0;
}
```

A LAGRANGE.C programot futtassuk le az alábbi adatokkal:

```
Lagrange interpoláció

Az alappontok száma: 4
Az alappontok növekvõ sorrendben

x[0] = 1
y[0] = 1

x[1] = 2
y[1] = 1.8

x[2] = 3
y[2] = 2.2

x[3] = 4
y[3] = 2.5

Az interpoláció helye: 2.5
Az interpolált érték : 2 031250
```


5.3.4. Aitken interpoláció

A Lagrange interpolációs polinom helyettesítési értékének kiszámítására gyakran használják az Aitken módszert. Ez az eljárás egyrészt egyszerűsíti a számítás elvégzését, másrészt módot ad arra, hogy - ha az interpoláció fokszáma nem bizonyul elegendőnek - egy újabb alappont felvételével a már kiszámolt helyettesítési értékből az eggyel magasabb fokszámú polinom helyettesítési értékét állítsuk elő. Az $x_0 \ldots x_n$ alappontokra támaszkodó n-edfokú interpolációs formula mindig előállítható két, ugyanezen alappontok egy részére támaszkodó n-1-ed fokú interpolációs polinommal.

$$y_{0,\ldots,n}(x) = \frac{y_{0,\ldots,n-1}(x)\ (x_n - x) - y_{0,\ldots,n-2,n}(x)\ (x_{n-1} - x)}{x_n - x_{n-1}}$$

ahol $y_{1,\ldots,n}(x)$ az 1,2...n alappontokra támaszkodó polinom. Könnyen belátható, hogy valóban n-edfokú polinom, másrészt az $x_0 \ldots x_n$ pont mindegyikében illeszkedik az eredeti függvényhez. Az Aitken interpoláció rekurzív módon állítja elő az egyre nagyobb fokszámú polinomokból adódó helyettesítési értéket.
Az iterációt rekurzív módon könnyen elvégezhetjük, ha az *y[n]* ordinátákat áttöltjük az *f[n]* tömbbe:

$$f_i = \frac{f_i(u - x_j) - f_j(u - x_i)}{x_i - x_j} \qquad (j = 1, 2, \ldots, n-1,\ i = j+1, \ldots, n-1, n)$$

$j \rightarrow$

$i \downarrow$ $\quad f_1, f_2, f_3, \ldots, f_n$ kiindulási értékek

$f_2', f_3', f_4', \ldots, f_n'$

$f_3'', f_4'', \ldots, f_n''$

$f_4''', \ldots, f_n'''$

$\ldots$

f_{n-1}^{n-1}, f_n^{n-1}

f_n^n

Például:

$$f_2' = \frac{f_2(u - x_1) - f_1(u - x_2)}{x_2 - x_1}$$

illetve

$$f_3'' = \frac{f_3'(u - x_2) - f_2'(u - x_3)}{x_3 - x_2}$$

kiszámítása.

Az iteráció befejeztével az u-hoz tartozó érték éppen az f_n^n lesz.

Az Aitken módszerrel működő interpolációs függvény:

```
double aitken(double *x, double *y, int n, double x1)
{
 double *f;
 int i,j;
   f = (double*) malloc((n)*sizeof(double));

   if ( x1 < x[0] || x1 > x[n-1]) return (0.0);
   for( i = 0; i < n; i++)
      f[i] = y[i];

   for( j = 0; j< n-2; j++)
   {
     for( i = j+1; i < n ; i++)
     {
     f[i] = ((x1-x[j])*f[i]-(x1 -x[i])*f[j])/(x[i] - x[j]);
     }
   }
   return (f[n-1]);
}
```

Az AITKEN.C programot futtassuk le az alábbi adatokkal:

```
Aitken interpoláció

Az alappontok száma: 4
Az alappontok növekvõ sorrendben

x[0] = 1
y[0] = 1

x[1] = 2
y[1] = 1.8
```



```
x[2] = 3
y[2] = 2.2

x[3] = 4
y[3] = 2.5

Az interpoláció helye : 2.5
Az interpolált érték  : 2.087500
```

5.3.5. Regresszió

A regresszió alapelve: az

$$x_i, y_i \qquad 1 \le i \le m$$

értékpárokra függvényt illesztünk úgy, hogy a

$$Q^2 = \sum_{i=1}^{m} (y_i - f(x_i))^2$$

mennyiség minimális legyen.
A minimumot az *f* függvény argumentumai szerinti differenciálhányadosok zérushelyei szolgáltatják.
Például az

$$f(x) = A + B \quad x$$

függvény esetén A és B a

$$\frac{\partial Q^2}{\partial A} = 0 \quad \text{és} \quad \frac{\partial Q^2}{\partial B} = 0$$

feltételekből határozhatók meg.

A valóságban a fenti deriváltak, illetve az *A* és *B* értékek lineáris esetben (regressziós egyenes) könnyen számíthatók. Ezért más közelítő függvény alkalmazása esetén, különböző transzformációkkal "lineárissá alakítjuk" a közelítendő függvényt, és ekkor már használhatjuk a regressziós egyenlet összefüggéseit. Az eredményt (A és B) természetesen szintén "vissza kell transzformálni". Ez könnyen belátható az exponenciális függvénnyel történő

közelítés segítségével. Az y tengely logaritmikus ábrázolása esetén egyenest kapunk ($\eta_i = \ln y_i$). A szükséges konstans megállapítása után a skálatényezőt visszaalakítjuk ($A = \exp(\alpha)$). Az alábbi táblázatban foglaltuk össze az ismert regressziós módszerek transzformációit és képleteit.

Típus	**Függvény**	ξ_i	η_i	**A**	**B**
Lineáris	$y = A + B\ x$	x_i	y_i	α	β
Exponenciális	$y = A\ e^{B\,x}$	x_i	$\ln(y_i)$	e^{α}	β
Hatványfüggvény	$y = A\ x^{B}$	$\ln(x_i)$	$\ln(y_i)$	e^{α}	β
Arrhenius	$y = A\ e^{-\frac{B}{x}}$	$-\frac{1}{x_i}$	$\ln(y_i)$	e^{α}	β
Reciprok	$y = \frac{1}{A + B\ x}$	x_i	$\frac{1}{y_i}$	α	β
Racionális tört	$y = \frac{A\ x}{1 + B\ x}$	x_i	$\frac{x_i}{y_i}$	$\frac{1}{\alpha}$	$\frac{\beta}{\alpha}$
Kvadratikus	$y = x\ (A + B\ x)$	x_i	$\frac{y_i}{x_i}$	α	β
Vektoriális	$y = \sqrt{A + B\ x^2}$	x_i^2	y_i^2	α	β
Hiperbola	$y = A + \frac{B}{x}$	$\frac{1}{x_i}$	y_i	α	β
Logaritmikus	$y = A\ \ln(B\ x)$	$\ln(x_i)$	y_i	β	$e^{\frac{\alpha}{\beta}}$

A táblázatban 10 fajta regresszió található. Mindegyik regresszióhoz megtaláljuk az $x_i = \xi_i$ és az $y_i = \eta_i$ transzformációkat. A transzformációk ismeretében kiszámítjuk az *S1*, *S2*, *S3*, *S4* és *S5* értékeket, melyből a β és az α meghatározható. Az *A* és *B* kiszámítható az α és a β ismeretében. A függvény akkor simul rá a pontsorozatra, ha a hiba minimális és a korreláció minél jobban megközelíti az 1-et.

$$S1 = \sum_{i=1}^{m} \xi_i \qquad S2 = \sum_{i=1}^{m} \eta_i \qquad S3 = \sum_{i=1}^{m} \xi_i^2$$

$$S4 = \sum_{i=1}^{m} \eta_i^2 \qquad S5 = \sum_{i=1}^{m} \xi_i\ \eta_i$$

$$\beta = \frac{m\ S5 - S2\ S1}{m\ S3 - S1^2} \qquad \alpha = \frac{S2 - \beta\ S1}{m}$$

Az α és β ismeretében az *A* és a *B* értéke meghatározható.

$$mean = \beta \left(S5 - \frac{S1\ S2}{m} \right)$$ négyzetes közép

$$dev = S4 - \frac{S2^2}{m}$$ eltérés (deviáció)

$$sd = dev - mean$$

$$Sdev = \frac{dev - mean}{m - 2}$$ átlagos eltérés

$$hiba = \sqrt{\left| \frac{sd}{m - 2} \right|}$$ átlagos hiba

$$Korr = \sqrt{\left| \frac{mean}{dev} \right|}$$ korrelációs együttható (≈ 1)

A REGRESS.C program futtatási eredménye:

```
Két paraméteres regresszió

Alappontok száma: 10
Alappontok

x[0] = 1
y[0] = 1.7

x[1] = 2
y[1] = 4.4

x[2] = 3
y[2] = 8.1

x[3] = 4
y[3] = 12.8

x[4] = 5
y[4] = 18.5

x[5] = 6
y[5] = 25.2

x[6] = 7
y[6] = 32.9

x[7] = 8
y[7] = 41.6

x[8] = 9
y[8] = 51.3

x[9] = 10
y[9] = 62.5

Az illesztés függvénytipusai

1.   Lineáris:          Y=A+B*X
2.   Exponenciális:     Y=A*E^(B*X)
3.   Hatvány:           Y=A*X^B
4.   Arrhenius:         Y=A*E^(-B/X)
5.   Reciprok:          Y=1/(A+B*X)
6.   Racionális tort:   Y=A*X/(1+B*X)
7.   Kvadratikus:       Y=X*(A+B*X)
8.   Vektoriális:       Y=SQRT(A+B*X^2)
9.   Hiperbolikus:      Y=A+B/X
10.  Logaritmikus:      Y=A*LN(B*X)
11.  Legjobb illesztés!
```



```
A függvény tipusszáma: 11
Az illesztés hibája: 1e-4

Az illesztés tipusa:  Kvadratikus:     Y=X*(A+B*X)

Standard deviáció: 0.000205
Standard hiba    : 0.014302
Korreláció       : 0.999961

Paraméterek értéke
A= 1.190000
B= 0.502727
```

5.4. Numerikus integrálás (numerikus kvadratúra)

A numerikus integrálás feladata, a

$$\int_a^b f(x)dx$$

határozott integrál kiszámítása. Akkor alkalmazzuk, ha

- a függvény pontokban ismert (mérési eredmények),
- a függvénynek nem létezik primitív függvénye, például:

$$\sqrt{1+x^3},\quad e^{x^2},\quad \frac{1}{\ln(x)},\quad \frac{\sin(x)}{x},$$

- bonyolult függvényt Lagrange-polinomjával közelítünk.

A numerikus integrálás formulája vagy más elnevezéssel a numerikus kvadratúra formulája:

$$\int_a^b y(x)dx = \sum_{i=0}^{n} c_i \quad y(x_i) + E(y)$$

A hiba

$$E(y) = \int_a^b y(x)dx - \sum_{i=0}^{n} c_i \quad y(x_i)$$

A numerikus integrálás célja az E(y) csökkentése.

Integrálási módszernek nevezzük a c_i súlyok és az x_i alappontok bizonyos megválasztási rendszerét. Integrálási módszer pontossági foka vagy a pontossági rendje n, ha legfeljebb n-edfokú polinomra teljesül a

$$E(y) = 0 .$$

Az E(y) csökkentésére két lehetőségünk van:

- Magasabb fokszámú közelítést alkalmazunk.
- Részintervallumokra bontjuk az integrált.

A Numerikus integrálási módszereket egyszerű és összetett kvadratúra módszerek csoportjába oszthatjuk. Az egyszerű kvadratúra módszerben az integrálás intervallumát egyetlen intervallumnak tételezzük fel és az integrálási módszer által meghatározott az x_i-k száma.

Az összetett integrálási módszerek esetében az integrálás intervallumát részintervallumokra osztjuk és ezekre a részintervallumokra alkalmazzuk az egyszerű kvadratúra módszereket. A teljes integrál a részintegrálok összege.

Mind az egyszerű, mind az összetett kvadratúra módszerek esetén a célunk az E hiba csökkentése. Ezt kétféleképpen érhetjük el. Magasabb pontossági rendű kvadratúra módszert választunk, vagy az integrálás intervallumát részintervallumokra osztjuk, azaz összetett kvadratúra módszert alkalmazunk. Mind a két esetben az egész integrálási intervallumra eső x_i-k száma megnő. A számítási időt elsősorban az x_i-k száma határozza meg.

Kérdés az, hogy ugyanakkora számítási idő mellett (ugyanannyi x_i) az egyszerű vagy az összetett kvadratúra a pontosabb. A részintervallumok számának növelésével majdnem minden függvényosztályra az E hiba gyorsan csökken, ugyanakkor az integrálási módszer nem válik bonyolulttá. Ezért a gyakorlatban majdnem mindig összetett kvadratúra módszert alkalmazunk. Bizonyítható, hogy azonos x_i szám mellett az összetett kvadratúra nemcsak egyszerűbb, de pontosabb is.

A numerikus integrálási módszerek osztályozásának egy másik szempontja, hogy az x_i alappontok elhelyezkedése:

- ekvidisztáns (egyenletes), vagy
- tetszőleges.

Táblázatos $y(x)$ megadás esetén nyilván célszerű az adott, legtöbbször ekvidisztáns osztást használni! Az ekvidisztáns lépésközzel egyszerűbb a számítás. Az x_i-k tetszőleges elhelyezése elvileg nagyobb pontosságot biztosít, a gyakorlatban azonban mégis többször használják az ekvidisztáns lépésközt. Egyenletes lépésközzel működik a Newton-Cotes módszer, speciális lépésközt használnak a Gauss, Gauss-Csebisev, Gauss-Laguerre és a Gauss-Hermite módszerek.

5.4.1. Newton-Cotes kvadratúra módszerek

A Newton-Cotes féle kvadratúra módszernél a függvényt egy *n*-ed fokú Lagrange-féle polinommal helyettesítjük. Az integrálásnál az *y(x)* függvényt az $l_i(x)\ \ y(x_i)$ Lagrange-féle interpolációs polinommal közelítjük. Mivel az integrálás és az összegzés felcserélhető, és az $y(x_i)$-k konstansok, ezért

$$\int_a^b y(x)dx$$

az integrál a következő alakra hozható:

$$\int_a^b \sum_{i=0}^{n} l_i(x)\ \ y(x_i)dx = \sum_{i=0}^{n} y(x_i)\ \ \int_a^b l_i(x)\,dx$$

ahol

$$l_i(x) = \prod_{\substack{j=0 \\ j\neq i}}^{n} \frac{x - x_j}{x_i - x_j}$$

Alkalmazva a

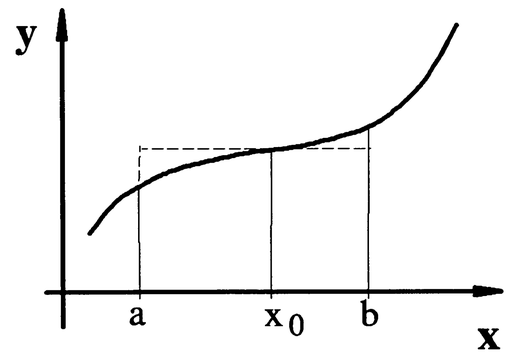
$$c_i = \int_a^b l_i(x)\,dx \quad \text{helyettesítést}$$

$$\int_a^b y(x)dx = \sum_{i=0}^{n} c_i\ \ y(x_i) \qquad (1)$$

(Az $l_i(x)$-k integrálása kis *n*-ek esetén egyszerű.)

5.4.1.1. Téglalap formula

A n = 0 esetén kapjuk a téglalap formulát, ilyenkor a görbe alatti területet téglalappal helyettesítjük. Ritkán szokás használni. Ilyenkor $l_i(x) = 1$



5 10 ábra Téglalap formula alkalmazása

$$\int_a^b y(x)\,dx \approx \sum_{i=0}^{0} y(x_0)\ \ \int_a^b 1dx = (b - a)\ \ y(x_0)$$

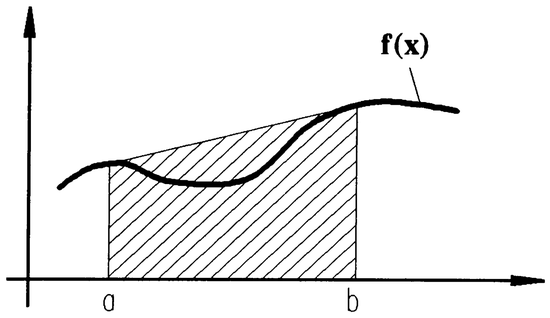
ahol x_0 tetszőlegesen választható, de gyakori az $x_0 = a$ (alsó téglalap), vagy az $x_0 = b$ (felső téglalap) választása. Összetett kvadratúra esetén minden x_i szerepel, kivéve az y(a) vagy az y(b).

A módszert használó programrészletet alkalmazzuk:

```
double teglalap(double a, double b, int n, fp f)
{
 int    i;
 double s, h;
   h = (b-a)/n;
   s = 0;
   for(i = 1; i < n; i++)
   {
     s +=  (*f)(a+h*i);
   }
   return h * ((*f)(a) + s + (*f)(b));
}
```

5.4.1.2. *Egyszerű (kis) trapéz formula*

Az n = 1 választás mellett a görbe alatti területet trapézzal helyettesítjük.



5 11 ábra Egyszerű trapéz formula alkalmazása

Az y(x) függvényt az elsőfokú Lagrange interpolációs polinommal közelítjük:

$$y(x) \approx \frac{x - x_1}{x_0 - x_1} y(x_0) + \frac{x - x_0}{x_1 - x_0} y(x_1)$$

a c_i együtthatók meghatározása:

$$c_0 = \int_a^b \frac{x - x_1}{x_0 - x_1} dx = \left[\frac{x^2}{2(x_0 - x_1)} - \frac{x\ x_1}{x_0 - x_1} \right]_a^b = \frac{b - a}{2}$$

$$c_1 = \int_a^b \frac{x - x_0}{x_1 - x_0} dx = \left[\frac{x^2}{2(x_1 - x_0)} - \frac{x\ x_0}{x_1 - x_0} \right]_a^b = \frac{b - a}{2}$$

ahol természetesen $x_0 = a$ és $x_1 = b$.

A kis trapéz formula a c_i együtthatók (1)-be helyettesítésével

$$\int_a^b y(x)\,dx \approx \frac{b - a}{2} y(x_0) + \frac{b - a}{2} y(x_1) = \frac{b - a}{2} \big(y(a) + y(b)\big)$$

és a

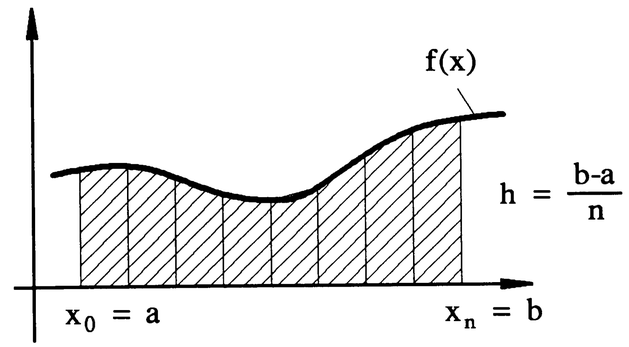
$$h = b - a$$

figyelembe vételével kapjuk meg az egyszerű trapéz formulát, amely egy h alapú, y(a) és y(b) párhuzamos oldalú trapéz területét adja:

$$\int_a^b y(x)\,dx \approx \frac{h}{2} \cdot \big(y(a) + y(b)\big)$$

5.4.1.3. *Összetett (nagy) trapéz formula*

Ha egy $[x_0, x_n]$ intervallum $h = \frac{x_n - x_0}{n} = \frac{b - a}{n}$ részintervallumaira alkalmazzuk az egyszerű trapéz formulát, akkor kapjuk meg az összetett vagy az ún. nagy trapéz formulát. Ekkor minden x_i az előző részintervallum végpontja és a következő kezdőpontja, kivéve az a,b pontokat, amelyek csak egyszer szerepelnek.



5 12 ábra Összetett trapéz formula alkalmazása

$$\int_a^b y(x)\,dx \approx h\left[\frac{y(a)}{2} + \sum_{i=1}^{n-1} y(x_i) + \frac{y(b)}{2}\right]$$

ahol

n	az osztások száma,
$h = \frac{b-a}{n}$	a konstans lépéstávolság,
$a = x_0$	alsó határ,
$b = x_n$	felső határ,

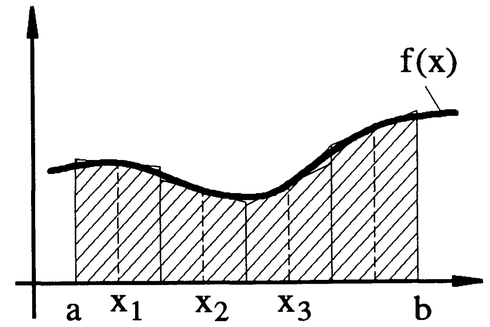
Megjegyzés: A h lépéstávolság kétszeres csökkentése a trapéz képlet hibáját kb. négyzetesen csökkenti Ebből következik, hogy a h és h/2 lépéstávolsággal kiszámított integrál értékek egybeeső jegyei helyesnek vehetők. (Ezt az összefüggést a számításunk ellenőrzésére használhatjuk.)

A módszert használó programrészletet alkalmazzuk:

```
double trapez(double a, double b, int n, fp f)
{
 int    i;
 double s, h;
   h = (b-a)/n;
   s = 0;
   for(i = 1; i < n; i++)
   {
     s += (*f)(a+h*i);
   }
   return h * ((*f)(a)/2 + s + (*f)(b)/2);
}
```


5.4.1.4. Érintő formula

Az elsőfokú közelítés egy másik lehetséges megoldása az érintő formulához vezet. Ekkor a függvényt az [a,b] tartomány közepén felvett alappontban az érintője egyenletével közelítjük.



5 13 ábra Érintő formula alkalmazása

Az összetett érintő formula:

$$\int_a^b y(x)dx \approx h\sum_{k=1}^{n} y(x_k)$$

ahol

$$h = \frac{b-a}{n}$$

$$x_k = a + \left(k - \frac{1}{2}\right) h \qquad k = 1,2, \ldots n$$

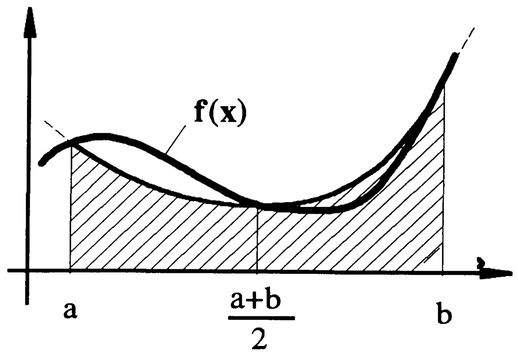
A közelítés hibája

$$E(y) = \frac{1}{12}\left(\frac{b-a}{2}\right)^2 y''_{max}(x)$$

vagyis kétszeres felbontásnál a hiba a negyedrészére csökken.

5.4.1.5. *Egyszerű (kis) Simpson formula*

A görbe alatti területet másodfokú polinomokkal (parabola) határolt trapézszerű alakzattal helyettesítjük.
A Newton-Cotes formulából n = 2 választás mellett,



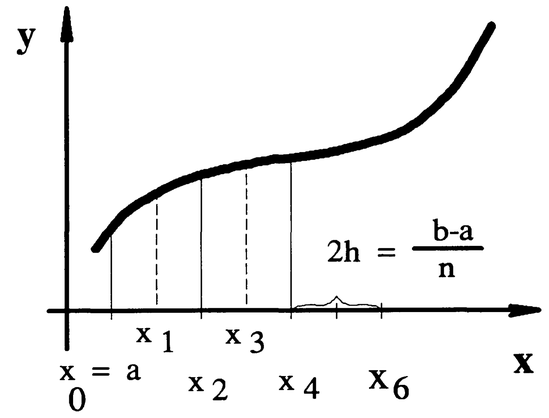
5 14 ábra Egyszerű Simpson formula alkalmazása

az eddigiekhez hasonlóan, bizonyítható (illetve a Lagrange-féle interpolációs polinomokból integrálással levezethető) a másodfokú (parabola) közelítést felhasználó kvadratúra képlet, amelyet az egyszerű vagy kis Simpson formula néven ismerünk:

$$\int_a^b y(x)\,dx \approx \frac{b-a}{6}\left[y(a) + 4\ y\left(\frac{a+b}{2}\right) + y(b)\right]$$

5.4.1.6. *Összetett (nagy) Simpson formula*

Az összetett Simpson formulát akkor kapjuk meg, ha *n* számú, *2h* hosszúságú intervallum szakaszra képezzük az összeget.



5 15 ábra Összetett Simpson formula alkalmazása

$$\int_a^b y(x)\,dx \approx \frac{2\ h}{6}\left[y(x_0) + 4y(x_1) + y(x_2) + y(x_2) + 4y(x_3) + y(x_4)\ldots\right] =$$

$$= \frac{h}{3}\left[y(x_0) + 4y(x_1) + 2y(x_2) + 4y(x_3) + 2y(x_4) +\right.$$

$$\left. +\ldots 2y(x_{n-2}) + 4y(x_{n-1}) + y(x_n)\right]$$

Tehát az összetett Simpson formula esetén:

- az a és b pontok egyszeres súllyal ($x_0 = a$, $x_n = b$),
- a páratlan sorszámúak 4-szeres súllyal,
- míg, a páros sorszámúak (mivel ezek a megelőző tartomány végei is és a következő tartomány első pontjai is) kétszeres súllyal rendelkeznek.

A Simpson formula igen nagy előnye, hogy egyszerűsége mellett nagy a pontossága. A Simpson formula hibája:

$$E(y) = \frac{1}{180}\left(\frac{b-a}{n}\right)^4 y^{IV}_{max}(x)$$

Beláthatjuk, ha a *h* lépéstávolságot kétszer kisebbre vesszük, akkor a Simpson képlet hibája kb. 16-odára csökken, s ezért a *h/2* lépéstávolsággal számított integrál értéke egy értékes jeggyel többet tartalmaz, mint a *h* lépéssel számított integrál értéke.

A módszert használó programrészletet alkalmazzuk:

```
double simpson(double a, double b, int n, fp f)
{
 double h, s;
 int    i;
   s = 0;
   h = (b-a)/n;

   for(i = 1; i < n; i++)
   {
     s = s+4*(*f)(a+i*h)*(i % 2)+2*(*f)(a+i*h)*((i+1) % 2);
   }
   s += (*f)(a)+(*f)(b);
   s *= h/3;
   return s;
}
```

Feladat: Számítsuk ki a

$$\int_0^1 \frac{4}{1+x^2}dx$$

integrál értékét.

Az INTEGRAL.C program fg1 függvényét módosítjuk:

```
/* Az fg1  a pi = 3.141592 értékét adja a [0,1] intervallumban */

double fg1(double x)
{
  return(4.0/(1.0+x*x));
}
```


Az INTEGRAL.C program futtatási eredménye:

```
Integrálás téglalap, trapéz és Simpson módszerrel
A program az f(x)=4./(1 +x*x) függvény határozott
integrálját számitja ki az adott intervallumban.
Javasolt adatok: alsó határ:0, felsõ határ:1, intervallum: 100

Alsó határ          : 0
Felsõ határ         : 1
Intervallumok száma: 100
A téglalap integrál : 3.171576
A trapéz integrál   : 3.141576
A Simpson integrál  : 3 141593
```

5.4.2. Romberg eljárás

Az *n* alappontra támaszkodó Simpson formula előállítható az *n* alappontos és az *n/2* alappontos trapéz formula lineáris kombinációjából:

$$S_n = \frac{4}{3}t_n - \frac{1}{3}t_{\frac{n}{2}}$$

Az s_n Simpson összegek jobban közelítik az integrált, mint a t_n trapéz összegek

$$s_n = t_n + \frac{1}{4^1-1}\left(t_n - t_{\frac{n}{2}}\right) \qquad n = 2, 4, 8,...$$

Ennek mintájára újabb képleteket állíthatunk elő, melyekről belátható, hogy egyre jobban közelítik az integrál értékét.

$$r_n = s_n + \frac{1}{4^2-1}\left(s_n - s_{\frac{n}{2}}\right) \qquad n = 4, 8, 16, ...$$

$$q_n = r_n + \frac{1}{4^3-1}\left(r_n - r_{\frac{n}{2}}\right) \qquad n = 8, 16, 32, ...$$

$$p_n = q_n + \frac{1}{4^4-1}\left(q_n - q_{\frac{n}{2}}\right) \qquad n = 16, 32, 64 ...$$

...

Az algoritmus rekurzívan hívható, amíg a legutolsó két közelítés elegendően megegyezik.

A számítás menete:

t_1

t_2 s_2

t_4 s_4 r_4

t_8 s_8 r_8 q_8

t_{16} s_{16} r_{16} q_{16} p_{16}

...

5.4.3. *Nem ekvidisztáns osztású kvadratúra: Gauss és Csebisev formulák*

Ha az alappontokat bizonyos szempontok szerint választjuk, akkor sokkal jobban közelítő interpolációs polinomokhoz juthatunk. Ha ezeket a polinomokat integráljuk, akkor jobban konvergáló integrálást végezhetünk. Természetesen itt is részintervallumokra osztunk és összetett kvadratúra képleteket használunk.
A Gauss és a Csebisev formulák a [-1,1] intervallumban történő integrálásra alkalmasak. Ha az

$$\int_a^b y(x)dx$$

integrálon elvégezzük az

$$x = \frac{b-a}{2}\xi + \frac{a+b}{2}, \qquad \eta(\xi) = \frac{b-a}{2}y(x)$$

transzformációt, akkor a

$$\int_{-1}^{1} \eta(\xi)\delta\xi$$

formájú integrálhoz jutunk.

Gauss kvadratúra:

$$\int_{-1}^{1} \eta(\xi)d\xi \approx \sum_{i=1}^{n} a_i \eta(\xi_i) \qquad n = 1, 2, \ldots$$

ahol az együtthatók és a ξ_i értékek a Legendre polinomokból számolhatók.

Csebisev kvadratúra:

$$\int_{-1}^{1} \eta(\xi)d\xi \approx \frac{2}{n}\left[\eta(\xi_1) + \eta(\xi_2) + \ldots \eta(\xi_n)\right] \qquad n = 2,3,4,5,6,7.$$

A ξ_i értékek a Legendre illetve a Csebisev polinomok gyökeiként határozhatók meg, például Csebisev polinom esetén:

n	ξ
1	0
2	$\pm \frac{1}{\sqrt{3}}$
3	0 és $\pm \frac{\sqrt{2}}{2}$
4	$\pm$ 0,1875924741 és $\pm$ 0,7946544723
5	0 és $\pm$ 0,3745414096, $\pm$ 0,8324974870
6	$\pm$ 0,2666354015, $\pm$ 0,4225186538 $\pm$ 0,8662468181
7	0 és $\pm$ 0,3239118105 $\pm$ 0,5296567753 $\pm$ 0,8838617008

F1. A Turbo Pascal és a Turbo C nyelv összehasonlítása

Ebben a fejezetben röviden áttekintjük a Turbo Pascal és a Turbo C közötti hasonlóságokat és különbségeket. Először bemutatjuk a program szerkezetét, a programozás elemeit és az adatszerkezeteket. Az összehasonlításokat példával illusztráljuk.

F1.1. A program szerkezete

Először a program szerkezete alapján vizsgáljuk meg a két nyelv közötti különbségeket.
A Pascal nyelvű program szerkezete a következő:

```
program ProgramNev;
uses <könyvtárak>;
< deklarációk kezdete
const <konstansok>;
type  <típusdefiníciók>;
var  <globális deklarációk>;
procedure Nev(paraméterek);
<lokális deklaráció>
begin
   <utasítások>
end;
function fnev(paraméterek):<típus>;
<lokális deklaráció >
begin
     <utasítások>
end;
. . .
deklaráció vége >

begin  { a főprogram blokkjának kezdete }

 < utasítások >

end. { a főprogram blokkjának vége }
```

A főprogram blokkjában lévő utasítások kerülnek végrehajtásra, ha azok eljárás- és függvényhívásokat is tartalmaznak, akkor az eljárás- és függvényblokk is végrehajtásra kerül. Minden azonosítót - konstans, típus, változó, eljárás és függvény - deklarálni kell.

A C program struktúrája sokkal szabadabb:

```
< preprocesszor parancsok >
< típusdefiníciók >
< függvény prototípusok >
< globális változók >
< függvények >
```

A függvények szerkezete a következő:

```
<típus> FuggvenyNev(<paraméterek>)
{
     < lokális deklaráció >
     < utasítások >
return <visszatérési érték>
}
```

A C-ben is minden nevet deklarálni kell. Valamelyik függvénynek kötelezően a **main** nevet kell adni, ez lesz a főprogram. Amikor a program végrehajtása megkezdődik, akkor a **main** függvény hívódik meg, és azon belül kerül sor a többi függvény aktivizálására. Néhány függvénynek a típusa **void**, ami azt jelenti, hogy nincs visszatérési értéke, így ezek hasonlóak a Pascal eljárásokhoz (**procedure**).

Az alábbi mintaprogramok illusztrálják a Pascal és a C nyelv program-szerkezete közötti hasonlóságot és a különbséget. A programban két függvényt aktivizálunk. A *csere* függvény két változó tartalmát cseréli fel, a *max* pedig két paramétere közül a maximálisat adja vissza. Értékeljük ki, és hasonlítsuk össze a két programot.

Turbo Pascal

```
program pelda1;
var
  I,J,K: integer;

function Max(A,B:integer):integer;
begin
   if A > B then  Max:= A
          else  Max:= B;
end; { Max függvény vége }

procedure Csere(var A,B:integer);
var
   Temp: integer;
begin
    Temp:= A; A:= B;
    B:= Temp;
end; { Csere procedure vége }
begin { programblokk eleje }
   I := 12; J := 20;
   writeln('Csere elõtt: I = ',I:2,
         ' J= ',J:2);
   Csere(I,J);
   writeln('Csere után: I = ',I:2,
         ' J = ',J:2);
   K:= Max(I,J);
   writeln(' Max. érték= ',K:2);

end. { programblokk vége }
```

Turbo C

```
#include <stdio.h>
int i,j,k;

int max(int a, int b)
{
  if (a > b) return a;
    else return b;
} /* max függvény vége */

void csere(int *a, int *b)
{
 int temp;

  temp= *a; *a= *b:
  *b= temp;
} /* csere függvény vége */
main()
{  i = 12;  j = 20;
   printf(" Csere elõtt:
     i= %2d j= %2d\n",i,j);
   csere(&i, &j);
   printf("Csere után:
     i= %2d j= %2d\n",i,j);
 k = max(i,j);
 printf(" Max. érték =
     %2d\n",k);
 return;
} /* a main függvény vége */
```

A Turbo C programban definiálhattuk volna az *i,j,k* változókat a **main** függvény belsejében lokálisan, azonban ehelyett a globális definíciót választottuk, hogy hasonló legyen a Pascal programhoz. A programban láthatunk megjegyzéseket. Hasonlítsuk össze az alábbi programrészleteket:

Turbo Pascal	**Turbo C**
`(* megjegyzés *)`	`/* megjegyzés */`
`(* ez is { jo megjegyzés } *)`	

F1.2. A programozás elemeinek összehasonlítása

Hasonlítsuk össze a programozás hat alapelemét: I/O műveletek, adattípusok, műveletek, feltételes műveletek, eljárások és függvények. Nézzünk ezekre Pascal és C nyelvű példákat.

F1.2.1. Az output művelet

A Pascal nyelvben a képernyőre való írás a **write** és a **writeln** eljárásokkal történik. A C nyelvben a *printf* függvény használjuk. A *printf* első argumentuma idézőjel (") között megadott ún. formátum-sztring. A formátum-sztring konverziós specifikációkat tartalmaz, amelyben a formátumvezérlő szekvenciát százalékjel (%) előzi meg. A *printf* függvény esetén a formátum-sztring tartalmaz olyan karaktereket is, amelyeket nem előz meg % jel. Ezeket egyszerűen megjeleníti a függvény.

```
printf( <formátum sztring>, <tétel>, <tétel>... );
```

ahol

<formátum sztring> sztring változó vagy sztring konstans, amely a megfelelő konverziót, előírást tartalmazza,

<tétel> kifejezések.

Táblázatban foglaltuk össze a leggyakrabban használt formátumokat:

d	előjeles decimális egész.
u	előjel nélküli decimális egész.
o	előjel nélküli oktális egész.
x	előjel nélküli hexadecimális egész.
c	egyetlen ASCII karakter.
s	karaktersorozat.
f	lebegőpontos szám, fixpontos alakban
e	lebegőpontos szám, exponenciális alakban

A kiírás után az új sor kezdését a formátum-sztring végén a '\n' szekvencia jelenti.

Turbo Pascal	Turbo C
`program pelda2:`	`#include <stdio.h>`
`var`	`#include <string.h>`
`  A,B,C: integer;`	`main()`
`  R   : real;`	`{  int a,b,c;`
`  Nev  : string[20];`	`   float r:`
`  CH   : char;`	`   char ch, nev[21];`
`begin`	
`  A:= 2; B:= 11; C:= -2;`	`  a= 2; b= 11; c= -2;`
`  C:= A+B;`	`  c= a+b;`
`  R:= 26.5;`	`  r= 26.5;`
`  Nev:= 'Zoltan';`	`  strcpy(nev,"Zoltan");`
`  CH:= 'i';`	`  ch= 'i';`
`  writeln('Pascal példák:');`	`  printf("C példák:\n");`
`  writeln;`	`  printf("\n");`
`  writeln('Mi a "kutya" neve?');`	`  printf("Mi a \"kutya\" neve?\n");`
`  writeln('''Bodri''',`	`  printf("\'Bodri\'  mondta`
`        '  mondta Laci.');`	`Laci.\n");`
`  writeln(A,' + ',B,'= ',C);`	`  printf("%d + %d = %d\n",a,b,c);`
`  writeln('Mennyi a súlya: ',`	`  printf("Mennyi a súlya: %6.2f`
`        R:6:2,' kg');`	`kg\n",r);`
`  writeln('Mi a neved: ',Nev);`	`  printf("Mi a neved: %s\n",nev);`
`  writeln('Válasz (i/n): ',CH);`	`  printf("Válasz (i/n):%c\n",ch);`
`  writeln('A = ',A:2);`	`  printf("a = %2d\n",a);`
`  writeln('A *  A = ',A*A:2);`	`  printf("a * a = %2d\n",a*a);`
`end.`	`}`

A C -ben használhatjuk a *puts* és a *putchar* kimeneti függvényeket is. A *puts* a paraméterében kapott sztringet kiírja a képernyőre A *putchar* még egyszerűbben működik, csak egyetlen karaktert ír ki.

Turbo Pascal	Turbo C
`writeln(Nev);`	`puts(Nev);`
`writeln('Rendben!'); writeln;`	`puts("Rendben!\n");`
`write(CH);`	`putchar(ch);`

F1.2.2. Adattípusok

Az alábbi táblázatban összefoglaltuk a két nyelv adattípusait:

Turbo Pascal			Turbo C		
Típus	**Hossza (byte) pontosság**	**Tartománya**	**Típus**	**Hossza (byte)**	**Tartománya**
char	1	0 - 255	**char**	1	-128 - 127
byte	2	0 - 255	**unsigned char**	1	0 - 255
integer	2	-32768-32767	**short**	2	-32768 - 32767
			int	2	-32768 - 32767
word	2	0 - 65535	**unsigned int**	2	0 - 65535
longint	4	$-2^{31} - (2^{31} - 1$	**long**	4	$-2^{31} - (2^{31} - 1)$
			unsigned long	4	$0 - (2^{32} - 1)$
single			**float**	4	±3.4e±38
real	6	1e-38-1e+38	-		
double	8	±1.7e±308	**double**	8	±1.7e±308
extended	10	±3.4e-4932-1.1e+4932	**long double**	10	±3.4e-4932-1.1e+4932
boolean	1	False, True	-		

Megjegyezzük, hogy nincs logikai adattípus a C nyelvben, azok az utasítások, amelyek logikai értéket várnak, a zérus értéket hamisnak, bármely más (pozitív vagy negatív) értéket pedig igaznak tekintik.

A Turbo C is támogatja a sorszámozott adattípust, amelyben felsorolt nevek a Pascal-hoz hasonlóan egész sorszámokat kapnak.

Turbo Pascal	Turbo C
type Napok = (hetfo, kedd, szerda, csutortok, pentek, szombat, vasarnap);	**enum** napok {hetfo, kedd, szerda, csutortok, pentek, szombat, vasarnap};
var Ma : napok;	**enum** napok ma;
begin Ma := vasarnap;	ma = vasarnap;

F1.2.3. Műveletek

A Turbo C-ben a szokásos műveleti jeleken kívül, további műveleti jeleket is találunk. A két nyelvben lényeges különbség van az értékadás művelete között. Pascal nyelvben az értékadás jele a := (ami utasítás), míg C-ben csak az = (operátorként szerepel).
Hasonlítsuk össze a két nyelvben használatos operátorokat:

Műveletek	Turbo Pascal	Turbo C
- elõjel	A:=-B;	a=-b;
+ elõjel	A:=+B;	a=+b;
logikai nem	not Flag;	!flag;
bitenkénti komplemens	A:=not B;	a=~b;
címe	A:=Addr(B);	a=&b;
indirekt hivatkozás	A:=IntPtr^;	a=*intptr;
mérete	A:=SizeOf(B);	a=sizeof(b);
inkrementálás (léptetés)	A:=Succ(A);	a++ és ++a
dekrementálás (léptetés)	A:=Pred(A);	a-- és --a
szorzás	A:=B*C;	a=b*c;
egészek osztása	A:=B div C;	a=b/c;
lebegőpontos osztás	X:=B/C;	x=b/c;
modulus képzés	A:=B mod C;	a=b%c;
összeadás	A:=B+C;	a=b+c;
kivonás	A:=B-C;	a=b-c;
biteltolás jobbra	A:=B shr C;	a=b >> c;
biteltolás balra	A:=B shl C;	a=b << c;
nagyobb, mint	A > B	a > b
nagyobb, vagy egyenlõ	A >= B	a >= b
kisebb, mint	A < B	a < b
kisebb, vagy egyenlõ	A <= B	a <=b
egyenlõ	A = B	a == b
nem egyenlõ	A <> B	a != b
aritmetikai és	A:= B and C;	a = b & c;
aritmetikai vagy	A:= B or C;	a= b | c;
aritmetikai kizáró vagy	A:= B xor C;	a= b ^ c;
logikai és	Flag1 AND Flag2	flag1 && flag2
logikai vagy	Flag1 OR Flag2	flag1 || flag2
értékadás	A:= B;	a = b;
	A:=A <operátor> B;	a <operátor>= b;

F1.2.4. Adatbevitel

Turbo Pascal programban az adatok olvasását a ***read()***, a ***readln()*** és a ***read(f,..)*** eljárások végzik. A Turbo C-ben a billentyűzetről jövő adatokat a ***scanf*** függvénnyel fogadjuk. A függvény paraméterei:

```
scanf(<formátum sztring>, <cím1>, <cím2>,...);
```

ahol

<formátum sztring> sztring, amelyben formátum jelzők vannak, hasonlóan mint a ***printf*** esetében,

<cím> cím, ahová a ***scanf*** tárolja a bejövő adatot. A változók címének megadásához a "címe" (&) operátort használjuk.

A C nyelvben gyakran használt függvény a ***gets()***, amely egy sztringet olvas addig, míg *<Enter>*-t nem ütünk és a ***getch()***, amely egyetlen karaktert olvas a képernyőn való megjelenítés nélkül.
Nézzünk néhány példát adatbevitelre:

Turbo Pascal	**Turbo C**
`A,B : integer;`	`int a,b;`
`X : single;`	`float x;`
`Nev : string[20];`	`char nev[21],ch;`
`CH : char;`	
`. . .`	`. . .`
`readln(A,B)`	`scanf("%d %d", &a, &b);`
`readln(Nev);`	`scanf("%s", Nev);` vagy
	`gets(nev);`
`readln(X,A);`	`scanf("%f %d",&x, &a);`
`readln(CH);`	`scanf("%c",&ch);`
`CH := readkey;`	`ch = getch();`

Jelentős különség van a sztring beolvasásánál a *scanf* és a *gets* függvények között. A *scanf* addig olvassa a karaktereket, míg nem talál üres helyet (*space*), tabulátort vagy új sort (*newline*). Ezzel szemben a *gets* minden karaktert beolvas, beleértve az üres helyet, a tabulátort és addig olvas, míg az *<Enter>* billentyűt nem nyomtuk le.

F1.2.5. A blokk utasítás

Mind a Pascal-ban, mind a C-ben megtaláljuk a blokk utasítás használatának koncepcióját, ami azt jelenti, hogy a blokk utasítás egyetlen utasítássá fogja össze az általa közrezárt utasításokat.

Turbo Pascal	**Turbo C**
`begin`	`{`
`< utasítás >;`	`< utasítás >;`
`. . .`	`. . .`
`< utasítás >`	`< utasítás >;`
`end;`	`}`

Bár a két megoldás formája nagyon hasonló, mégis három fontos különbség van közöttük:

- Turbo Pascal-ban nem kell pontosvesszőt (;) tenni az utolsó `<utasítás>` után, mint a Turbo C-ben, ahol kell tennünk.
- A C-ben sosem teszünk pontosvesszőt a blokkzáró (}) jele mögé, azonban a Turbo Pascal-ban tennünk kell.
- A C-ben blokk tartalmazhat lokális deklarációt is, míg a Pascal blokkban csak utasítások helyezhetők el.

F1.2.6. Feltételes utasítások

Mind a Turbo Pascal-ban, mind a Turbo C-ben kétfajta feltételes végrehajtási konstrukció használható: az **if** (**then**) **else** utasítás és a **case**, ill. **switch** utasítás.
Az **if** (**then**) **else** nagyon hasonló:

Turbo Pascal	**Turbo C**
`if <logikai kifejezés>`	`if <kifejezés>`
`then <utasítás>`	`<utasítás>;`
`else <utasítás>;`	`else <utasítás>;`

A C-ben és a Pascal-ban is az **else** feltétel opcionális, és az <utasítás> blokk utasítás is lehet.
A feltételes utasítás közötti különbségeket az alábbiakban foglaltuk össze:

- C-ben a `<kifejezés>` tetszőleges kifejezés lehet, amelynek zérus értéke esetén hamis, nem zérus esetén pedig igaz elágaztatás történik.
- A C-ben a `<kifejezés>`-nek zárójelben kell lennie.
- A C-ben nincs **then**.
- A C-ben ki kell tenni az utasítás után a pontosvesszőt, kivéve ha blokk utasítás van helyette.

Nézzünk meg néhány példát a feltételes utasításokra:

Turbo Pascal	Turbo C

`if X = 0`
`  then writeln('X határozatlan')`
`  else begin`
`  X:= A div B;`
`  writeln('X = ',X);`
`  end;`
 	
`C:= A-B;`
`if C <> 0`
`  then C:= C+B`
`  else  C:=A;`
	`if (x == 0)`
	`  puts("x határozatlan");`
	`  else {`
	`  x = a/b;`
	`  printf("x = %d\n",x);`
	`  }`
	
	`if((c= a-b) != 0)`
	`  c += b;`
	`else c = a;`

A többirányú elágaztatás a Pascal-ban (**case** utasítás) és a C-ben is (**switch** utasítás) egyaránt megtalálható, van azonban néhány különbség közöttük:

Turbo Pascal	Turbo C

`case <kifejezés> of`
`  <lista>  : <utasítás>;`
`  <lista>  : <utasítás>;`
 	
`  <lista>  : <utasítás>;`
`  else <utasítások>`
`end;`
	`switch (<kifejezés>) {`
	`  case <tétel> : <utasítások>`
	`  case <tétel> : <utasítások>`
	
	`  case <tétel> : <utasítások>`
	`  default : <utasítások>`
	`}`

A Pascal-ban a *<lista>* értékek sorozata, amely tartalmazhat résztartományt is (például: 'A'..'Z') is. Azonban a C-ben a *<tétel>* csak egyetlen érték lehet.

Mindkét nyelvben az értékek a sorszámozott típusokra vannak korlátozva, és konstansok (egészek, karakterek és sorszámozott adatelemek) lehetnek.

További lényeges különbség van az *<utasítások>* megadása és feldolgozása között. A Pascal-ban a **case**-en belül egyszerű vagy blokk utasításokat szerepeltethetünk az adott esetben kapcsolódóan. Ezen utasítások közül legfeljebb egy kerül végrehajtásra.

A C nyelvben az egyes esetekhez kapcsolódóan több utasítást is megadhatunk. A megfelelő eset (**case**) kiválasztása után az összes következő esetekhez megadott utasítások is végrehajtódnak. A Pascal-szerű működés

megvalósításához az esethez tartó utasítássorozatot **break** utasítással kell bezárni.

Nézzünk egy példát a többirányú elágaztatásra, ahol láthatóan a Pascal tömörebben tudja a tételeket leírni.

Turbo Pascal	Turbo C

`case Ma of`
`hetfo..pentek:`
`  writeln('Dolgozni megyek.');`
 	
`szombat, vasarnap:`
`  begin`
`  if Ma = szombat then`
`  writeln('Moziba megyek.')`
`  else`
`  begin`
`  write('Ha meleg van, ');`
`  writelen('napozni fogok!');`
`  end;`
`  end;`
`else  writeln('Hibás adat!');`
 	
`end;`
	`switch(ma)`
	`{`
	`  case hetfo:`
	`  case kedd:`
	`  case szerda:`
	`  case csutortok:`
	`  case pentek:`
	`  printf("Dolgozni megyek.\n");`
	`  break;`
	`  case szombat:`
	`  case vasarnap:`
	`  if (ma == szombat)`
	`  puts("Moziba megyek!");`
	`  else`
	`  {`
	`  printf("Ha meleg van, ");`
	`  printf("napozni fogok!\n");`
	`  }`
	`  break;`
	`  default: puts("Hibás adat!");`
	`  break;`
	`}`

Megjegyezzük, hogy a C példában a **switch** utasítás **case** *<tétel>* részében nem lehet felsorolás, azért minden tételt külön kell kezelni, *hetfo*-től *pentek*-ig, itt adjuk meg a címkékhez tartozó utasítást. A **break** utasítás a **switch** utasítást követő utasításra adja a vezérlést. A program hasonlóan viselkedik a hétvége kiválasztása esetén is.

F1.2.7. Ciklusutasítások

A C nyelvben a Pascal nyelvhez hasonlóan háromfajta ciklusutasítás van: a **while**, a **do...while** és a **for**, melyeknek a Pascal nyelvben a **while**, a **repeat...until** és a **for** ciklusok feleltethetők meg.

F1.2.7.1. A while ciklus

A három ciklus közül legjobban a **while** ciklus hasonlít a két nyelvben, melyek formája a következő:

Turbo Pascal	Turbo C
while *<logikai kifejezés>* **do** *<utasítás>*,	**while** *(<kifejezés>)* *<utasítás>;*

Mindkét nyelvben blokk utasítást használunk, ha a ciklusban egynél több utasítást szeretnénk végrehajtani. A különbség a két nyelv között az, hogy a C sokkal rugalmasabban fogadja el a *<kifejezés>*-t. Példaként hasonlítsuk össze a két ciklust:

Turbo Pascal	Turbo C

`Ch := readkey;`
while `Ch <> 'q' do begin`
`  write(Ch); Ch := readkey;`
end;
	while`((ch = getch() ) != 'q')`
	`  putchar(ch);`

F1.2.7.2. A do .. while ciklus

A **do** ... **while** ciklus hasonlít a Pascal **repeat** ... **until** ciklusához. A formátumuk a következő:

Turbo Pascal	Turbo C
repeat *<utasítások >* **until** *<logikai kifejezés>;*	**do** *<utasítások>;* **while** *(<kifejezés>);*

Két jelentős különbség van a ciklusok között:

- A **do**...**while** ciklus addig hajtódik végre, amíg a kifejezés igaz, a **repeat**...**until** ciklus éppen ellenkezőleg működik, mert addig hajtódik végre, amíg a logikai kifejezés hamis.
- A **repeat**...**until** ciklusutasításnál nem kell a blokk utasítás abban az esetben sem, ha a ciklusmag több utasításból áll, míg a **do**...**while** esetén szükséges.

Turbo Pascal	Turbo C

repeat
`  write("A értéke: ");`
`  readln(A);`
until `(A >= -5) and (A <= 5);`
	do {
	`  printf("a értéke: ");`
	`  scanf("%d",&a);`
	} **while** `(a < -5 || a > 5);`

A másik jelentős különbség a C és a Pascal között, hogy a C-ben magasabb precedenciája van a relációs operátoroknak (>, <, stb.), mint a logikai operátoroknak (&&, ||). Ez azt jelenti, hogy nem szükséges zárójelezni a relációs kifejezéseket, mint ahogy azt a Pascal-ban tesszük.

F1.2.7.3. A for ciklus

A **for** ciklus mutatja a legnagyobb különbséget a Pascal és a C között. Pascal-ban a **for** ciklus kötött szerkezetű, míg C-ben túl rugalmas, megenged olyan konstrukciót is, amely által szinte elveszti az összes hasonlóságát a Pascal **for** ciklusával.

Turbo Pascal	Turbo C
for `<index>:=<kezdõ>` **to** **downto** `<végérték>` **do** *<utasítás>;*	**for** `(<kif1>; <kif2>; <kif3> )` *<utasítás>;*

A C-ben (mint ahogy ez a Pascal-ra is igaz), a **for** utasítás a **while** utasítás speciális esete:

```
<kif1>
while <kif2> {
  <utasítás>;
  <kif3>;
}
```

A *<kif1>* kezdőérték a ciklus inicializálására, a *<kif2>* a ciklus végértékének tesztelésére és a *<kif3>* a léptetés megadására szolgál.

Nézzünk meg néhány példát, amelyben a Pascal **while** ciklust használ:

Turbo Pascal	Turbo C

for J:= 1 **to** 12 **do begin**
writeln('J = ',J:2);
writeln('J*J = ', J*J:4);
writeln('J**3 = ',J*J*J:6);
end;
	for(j= 1; j <= 12; j++) {
	printf("j = %2d\n",j);
	printf("j*j = %4d\n",j*j);
	printf("j**3 = %6d\n",j*j*j);
	}

I:= 15; X:= I;
while (I > -100) **do begin**
X:=X + I;
writeln('X = ',X:4,' I = ',I:4);
I:=I-15;
end;
	for(i= 15, x= i; i >- 100; i-=15)
	{
	x += i;
	printf("x = %4d i = %4d\n",x,i);
	}

D:= 5; Y:=D/2.0; **while**(abs(Y*Y-D) >0.07) **do** Y:= (Y+D/Y)/2.0;	d = 5; **for**(y= d/2; fabs(y*y-d)>0.07; y= (y+d/y)/2.0) ; /* üres utasítás */

Az utolsó C példában a ciklus törzse üres utasítás, a feladatot a ciklus fejlécében oldottuk meg.

F1.2.8. Alprogramok

A Pascal-ban és a C-ben is vannak alprogramok. A Pascal az eljárásokat (**procedure**) és a függvényeket (**function**), míg a C csak a függvényeket ismeri.

A C függvényeket azonban deklarálhatjuk **void** típusúnak is, amely azt jelenti, hogy nem adnak vissza értéket

A Pascal függvényeknél a (formális) paraméterek megadása:

(<parnév>:<típus> ; <parnév>:<típus>; ...)

A C függvényeknél a paraméterek megadása:

(<típus> <parnév>, <típus> <parnév>, ...)

A függvény általános formája:

Turbo Pascal	Turbo C

function
Fnev(<paraméterek>):<típus>;
<lokális deklaráció>
begin
<utasítások>
end;
	<típus> Fnev(<paraméterek>)
	{
	<lokális deklaráció>
	<utasítások>
	}

Jelentős különbség van a két függvény között a visszatérési érték (függvényérték) megadásában.

Turbo Pascal	Turbo C

function *Min(A,B* : integer): integer;
begin
if A < B
then Min:= A
else Min:= B;
end;
	int min(**int** a, **int** b)
	{
	if (*a* < *b*)
	return *a*,
	else return *b;*
	}

A példában látható, hogy a függvényértéket a C függvény a **return** utasításban, míg a Pascal függvény a függvény nevéhez rendeli hozzá (:=).

A Pascal-ban használhatunk változó (**var**) paramétereket. A C nyelv azonban csak az érték szerinti paraméterátadást ismeri. Ha egy változó paramétert kívánunk átadni, akkor átadjuk annak a címét, ebben az esetben a formális paramétert pointernek kell definiálni.

A *Csere* függvénynek két cím szerint átadott paramétere van, mivel az eljárás mindkét paraméterbe ír, mivel felcseréli a tartalmukat.

Turbo Pascal	Turbo C

procedure Csere(**var** X,Y : real);
var
T: real;
begin
T:= X;
X:= Y;
Y:= T;
end;
	void csere(**double** *x,**double** *y)
	{
	double t;
	
	t = *x;
	*x = *y;
	*y = t;
	}

Függvények hívása á következő módon történik:

Turbo Pascal	Turbo C
`A:=2.5;`	`a= 2.5;`
`B:=4 8;`	`b= 4.8;`
`writeln('A = ',A,' B = ',B);`	`printf("a = %lf b = %lf\n",a,b);`
`Csere(A,B);`	`csere(&a,&b);`
`writeln('A csere után:');`	`printf("A csere után: \n");`
`writeln('A = ',A,' B =',B);`	`printf("a = %lf b = %lf\n",a,b);`

A C példában a cím operátort (&) használtuk, mikor átadtuk a *csere* függvénynek az *a* és *b* változókat.

F1.2.9. C függvények prototípusa

Jelentős különbség van a Pascal és a C nyelvek között a függvényhívások ellenőrzésében. A Pascal mindig ellenőrzi a függvények hívásánál a paramétereket, hogy rendre megegyeznek-e a függvényben deklarált paraméterek darabszámával és típusával. Tekintsük az alábbi Pascal függvényt:

```
function Max(I,J : integer): integer;
```

és hívjuk meg a függvényt valós értékkel:

```
A:= Max(4, 8.3);
```

Ebben az esetben a fordításkor a fordító típushibát jelez a 8.3 paraméterre, mivel a valós érték nem fogadható el egész típus helyettesítésére.

A C-ben másképpen van. Alapértelmezés szerint a C fordító nem végez hibavizsgálatot a függvényhívásnál. Nem ellenőrzi a paraméterek számát, sem típusát, sőt a függvény visszatérési értékének a típusát sem. Ennél fogva a C nyelvben hívhatunk olyan függvényeket is, amelyeket még nem definiáltunk. A fordító rugalmassága miatt elkövethető programozási hibák kiküszöbölhetők a függvények deklarációjának (prototípusainak) megadásával. Ez a megoldás hasonlít a Pascal-ban a **forward** deklarációhoz. Általában a használni kívánt függvények prototípusát a forrás file elejére gyűjtjük össze.

A függvény prototípusának deklarációja:

<típus> `Fnév(`*<típus> <pnév>, <típus> <pnév>, ...*`);`

Ez hasonló ahhoz, ahogy a Pascal deklarálja a függvényeket, de itt is van egy kis különbség. Vesszőt (nem pontosvesszőt) használunk a paraméterek

elválasztására, és az azonos típusú paraméterek esetén sem alkalmazhatjuk a listaszerű felsorolást.

Nézzünk néhány példát a prototípusok megadására.

```
int min(int a, int b);
void csere(double *x, double *y);
void csere_tetel(index *i, index *j);
void rendez(lista w, int n);
void kiir(lista w, int n);
```

Ha a függvények prototípusát akarjuk használni, érdemes a függvény prototípusait elkülöníteni egy fejléc (*include*) file-ban és a függvények fejléc file-ját *include*-ként (*#include*) betenni az összes olyan állományba, ahol ezeket a függvényeket használjuk.

F1.3. Az adatstruktúrák áttekintése

Ebben a fejezetben összehasonlítjuk a Pascal és a C adatstruktúráit. Bemutatjuk a mutatók, a tömbök, a sztringek, a struktúrák és a unionok használatát.

F1.3.1. Mutatók

C programban gyakrabban használjuk a mutatókat, mint a Pascal programban. Ez azért van, mivel a C nyelv a függvények hívásánál csak az értékszerinti paraméterátadást használja. Amennyiben függvényen belül kívánunk egy változónak értéket adni, akkor argumentumként a változó címét (mutatóját) kell átadni a függvénynek.

Néhány példán keresztül hasonlítsuk össze a mutatók deklarációját a két nyelvben.

Turbo Pascal	Turbo C
	`#include <alloc.h>`
type	
`intarray = array[0..10] of integer;`	
`pnode = ^node;`	
`node = record`	`typedef struct ns {`
`data : integer;`	`int data;`
`next : pnode;`	`struct ns *next;`
`end;`	`} node;`
var	`main()`
	`{`
`intptr : ^integer;`	`int *intptr;`
`buff1 : ^intarray;`	`int (*buff1)[11];`
`buff2 : array[0..20] of ^integer;`	`int *buff2[21];`
`phead : ^node;`	`node *phead;`
`head : node;`	`node head;`
`i : integer;`	`int i;`
begin	
`new(intptr);`	`intptr = malloc(sizeof(int));`
`new(buff1);`	`buff1 = malloc(sizeof(int)*11);`
`for i:=0 to 20 do`	`for(i=0; i<21; i++)`
`new(buff2[i]);`	`buff2[i]=malloc(sizeof(int));`
`new(phead);`	`phead=malloc(sizeof(node));`
`intptr^ := 32;`	`*intptr = 32;`
`buff1^[5] := 12;`	`(*buff1)[5] = 12;`
`buff2[3]^ := 10;`	`*buff2[3] = 10;`
`head.next := nil;`	`head.next = NULL;`
`head.data := 21;`	`head.data = 21;`
`phead^.next :=nil;`	`phead->next = NULL;`
`phead^.data :=21;`	`phead->data = 21;`
`dispose(intptr);`	`free(intptr);`
`dispose(buff1);`	`free(buff1);`
`for i:=0 to 20 do`	`for(i=0; i<21; i++)`
`dispose(buff2[i]);`	`free(buff2[i]);`
`dispose(phead);`	`free(phead);`
end.	`}`

A példában a *buff1* egy egész típusú tömbre mutató pointer, a *buff2 pedig* egy pointer tömb, melyeknek elemei egész típusra mutatnak. A *Head* egy **record** (C-ben pedig **struct**), *data* és *next* mezőkkel rendelkezik. A *Phead a head* típusú adatra mutató pointer.

F1.3.2. Tömbök

A tömbök sokkal egyszerűbben definiálhatók a C nyelvben, mint a Pascal-ban. A C-ben a tömböknek lehet egész, karakter vagy sorszámozott típusú indexei, míg a Pascal megenged bármilyen sorszámozott típust. C nyelvben a tömbindex határa 0-tól *(n-1)*-ig tart (ahol *n* a tömb mérete). A Pascal-ban a tömb kezdő- és végindexét tetszőlegesen választhatjuk.

A C nyelvben az egydimenziós tömbök és az egyszeres indirektségű mutatók között teljes az analógia. Ennek megfelelően az a[i] és a *(a+i) hivatkozás egyenértékű. Turbo Pascal for Windows 7.0-tól kezdődően a {$X+} fordítás esetén ugyancsak használhatjuk ezt az analógiát karakteres 0-kezdőindexű tombökre:

```
{$X+}
var  PT : array[0..10] of char;
 ...
     PT[2] := 'A';      vagy   (PT+2)^ := 'A';
```

A tömb általános formája:

Turbo Pascal	Turbo C
`<név> : array[<alsó>..<felsõ>] of <típus>;`	`<típus> <név>[<méret>];`
`<név> : array[<alsó>..<felsõ>] of <típus>;`	`<típus> <név>[<méret>];`

A többdimenziós tömb C-ben való deklarálása a következő:

<típus> <név>[<méret1>] [<méret2>] [<méret3>];

Eltérően a Pascal nyelvtől az indexeket dimenziónként külön zárójelben kell megadni: *tomb[1][1][1]*

A tömb használatára nézzük meg az alábbi *amin* függvényt, amelynek két paramétere van, az **int** típusú tömb és a tömb elemeinek száma, a visszatérési értéke pedig a tömb legkisebb eleme:

```
int amin(int a[], int n)
{
  int min = a[1], i;
  for (i= 2; i < n; i++)
     if(a[i] < min) min = a[i];
  return min;
}
```

F1.3.3. Sztringek

A szabványos Pascal nem tartalmaz külön adattípust a szövegek tárolására. A Turbo Pascal azonban operátorokkal, függvényekkel és eljárásokkal is segíti a sztring felhasználását.

A C (beleértve a Turbo C-t) nem definiált külön sztring adattípust. A szövegek tárolására **char** típusú tömböket használhatunk.

Turbo Pascal	**Turbo C**
`<név> : string[<méret>];`	`char <név>[<méret>];`
`type`	
`  MaxStr = string[255];`	`typedef char maxstr[256];`
	`typedef char *strptr;`
`var`	
`  Nev : string[80];`	`char nev[81];`
`  Puffer : MaxStr;`	`maxstr puffer;`
`  Cim : string[40];`	`char cim[41];`
	`strptr ptr;`

A két nyelv eltérő módon tárolja a sztring hosszára vonatkozó információt. Turbo Pascal-ban a deklaráció:

```
S : string[N];
```

amely megegyezik

```
S : array[0 .. N] of char;
```

Ebben a deklarációban a sztring hosszát *N* értéke határozza meg (max. értéke 255 lehet). Az aktuális hossz az *S[0]* elemében kerül tárolásra, a sztring karakterei pedig az *S[1]* elemétől helyezkednek el. A Pascal-ban a sztring típusú változóknak közvetlenül adhatunk értéket.

Turbo C-ben a sztring deklarációja:

```
char strarray[N];
```

vagy

```
char *strptr;
```

Az első deklarációban egy *N* elemű karakter tömbben kívánjuk a sztringet tárolni. A terület kezdőcímét *strarray* tartalmazza. A második deklarációban az *strptr* karakter típusra mutató pointer.

A C-ben a sztring hosszát nem tárolja a rendszer, hanem ehelyett a sztring végét jelzi egy 0 értékű byte-tal. Ez még helyet igényel a sztring vége után ('\0'), - a sztring karakterei a *strarray[0]* elemtől helyezkednek el.
A sztring valójában csak N-1 hosszú sztringet képes tárolni, hiszen 1 byte szükséges a sztring vége byte tárolására.

C-ben a sztringeknek közvetlenül nem tudunk értéket adni, erre a célra könyvtári függvények állnak rendelkezésünkre, mint például az ***strcpy***:

```
strcpy(strarray,"Ma nincs hideg!");
```

Azonban közvetlenül olvashatunk be sztringet a *scanf* vagy a *gets* függvényekkel.

A másik deklarációval gondosabban kell eljárnunk, mivel az *strptr* egy karakterre mutató pointer, amely alaphelyzetben sehová sem mutat, így a sztring számára helyet kell foglalnunk.
Helyfoglalás elvégzésére a ***malloc*** függvényt használjuk:

```
strptr = (char*) malloc(N*sizeof(char));
```

A ***malloc*** függvény segítségével *N* byte-ot foglalunk le a sztring számára és a címét beletesszük az *strptr* pointer változóba. Ezután már használhatjuk az ***strcpy*** függvényt a sztring feltöltésére. A C nyelvnek megfelelő string - kezeléssel a Turbo Pascal nyelvet is ellátták. A Turbo Pascal for Windows 1 0 verziótól kezdődően (pchar típus, strings *unit*).
Példa a karakterre mutató pointer használatára:

Turbo Pascal	**Turbo C**
`program sztring;`	`#include <stdio.h>`
`uses Strings;`	`#include <stdlib.h>`
`type StrPtr = Pchar;`	`typedef char* StrPtr;`
	`main()`
	`{`
`var Ptr : StrPtr;`	`StrPtr Ptr;`
`begin`	
`  GetMem(Ptr,80);`	`  Ptr = (StrPtr)malloc`
	`  (80*sizeof(char));`
`  strcopy(Ptr,'Ez egy teszt.');`	`  strcpy(Ptr,"Ez egy teszt.");`
`  writeln('Ptr = ',Ptr);`	`  printf("Ptr =%s\n",Ptr);`
`  FreeMem(Ptr,80);`	`  free(Ptr);`
`end.`	`}`

F1.3.4. Struktúrák

A Pascal és a C is megengedi a különféle típusú adatok együttes kezelését. Ezt valósítja meg Pascal-ban a **record**, C-ben pedig a **struct** típus, melyek formátuma a következő:

Turbo Pascal	Turbo C

type
  <rnév> = **record**
  <fnevek> : <típus>;
  <fnevek> : <típus>;
  . . .
  <fnevek> : <típus>;
end;
var
  <vnevek> : <rnév>;
	typedef struct {
	  <típus> <fnevek>;
	  <típus> <fnevek>;
	  . . .
	  <típus> <fnevek>;
	} <rnév>;
	
	<rnév> <vnevek>;

A C struktúrája és a Pascal rekordja:

Turbo Pascal	Turbo C

var
  <vnevek> : **record**
  <fnevek> : <típus>;
  <fnevek> : <típus>;
  . . .
  <fnevek> : <típus>;
end;
	struct <rnév> {
	  <típus> <fnevek>;
	  <típus> <fnevek>;
	  . . .
	  <típus> <fnevek>;
	} <vnevek>;

A struktúránál az <rnév> opcionális, csak akkor tegyük oda, ha azt tervezzük, hogy esetleg más változó <rnév> típussal kívánjuk definiálni.

Nézzünk példát rekord és struktúra használatára:

Turbo Pascal	Turbo C

type
  Tanulo = **record**
  Veznev : string[25];
  Knev : string[20];
  Kora : integer;
  Magassaga: real;
  Jegyei : array[0..4] of integer;
end;
	struct tanulo {
	  char veznev[26];
	  char knev[21];
	  int kora;
	  double magassaga;
	  int jegyei[5];
	}fiu;

Turbo Pascal (folytatás)	Turbo C (folytatás)

var
  Fiu : Tanulo;
begin
  Fiu.Veznev := 'Kiss';
  Fiu.Knev := 'Viktor';
  Fiu.Kora := 12;
  Fiu.Magassaga:= 158.5;
  Fiu.Jegyei[1]:=5;
  Fiu.Jegyei[2]:=4;
end.
	main()
	{
	  strcpy(fiu.veznev,"Kiss");
	  strcpy(fiu.knev,"Viktor");
	  fiu.kora = 12;
	  fiu.magassaga = 158.5;
	  fiu.jegyei[1] = 5;
	  fiu.jegyei[2] = 4;
	}

A Pascal-ban a **with** utasítással rövidebben lehet a **record** mezeire hivatkozni, amelyhez hasonlót nem találunk a C nyelvben:

```
with Fiu do
begin
    Veznev:= 'Kiss';
     Knev := 'Viktor';
end;
```

A C nyelvben mindig ki kell írni a struktúra nevét, viszont van adattagot elérő (*member access*) operátora (->), is amit akkor használunk, amikor az operátornak a bal oldalán olyan mutató szerepel, amely a struktúrára mutat. Ha például a *ptanulo* a *tanulo* struktúrára mutat, akkor

```
strcpy(ptanulo -> veznev,"Kiss");
```

F1.3.5. Unionok

A Pascal és a C nyelv hasonlóan támogatják a union elvet: Pascal-ban szabadon variálható rekordban, a C-ben pedig **union** néven.
A megadás formátuma a következő:

Turbo Pascal	Turbo C

type
  <unév> = **record**
  <mezõlista>
  case <típus> **of**
  <vlista> : (<mezõlista>);
  <vlista> : (<mezõlista>);
  . . .
  <vlista> : (<mezõlista>);
end;
	union <unév> {
	  <típus> <fnév>;
	  <típus> <fnév>;
	  . . .
	  <típus> <fnév>;
	};

A Pascal-ban a <mezőlista> a szokásos **record** mezők, ahol az <fnevek> és a <típus> ismétlődnek.

Nézünk erre egy példát (habár nem teljesen azonosak):

Turbo Pascal	Turbo C
type	**typedef union**{
szo = **record**	**int** w;
case integer **of**	**struct** {
0 : (w : integer);	**char** lo;
1 : (lo,hi: byte);	**char** hi;
end;	} b;
var xp: szo;	} szo;
	szo xc;

A mezőkre való hivatkozás Pascal-ban `xp.w`, `xp.lo`, `xp.hi` és C-ben `xc.w`, `xc.b.lo` és `xc.b.hi`.

F1.4. Programozásbeli különbségek

F1.4.1. Betű-érzékenység

A Pascal nem érzékeny a kis- és nagybetűkre. Pascal-ban az index, Index, INDEX ugyanazt a változót jelenti, a C pedig három különböző változónak értelmezi.

F1.4.2. Típuskonverziók (type-casting)

A Pascal-ban csak korlátozott típuskonverziós előírás használható. Az Ord() függvény bármely sorszámozott típusról *integer*-re alakít, a Chr() *integer*-ről *char* típusra. A Turbo Pascal megenged ugyan néhány típus átalakítást (*integer*, *char*, *boolean*, stb., de a C nyelvben sokkal szabadabban térhetünk át egyik típusról a másikra, az eredmény azonban nem mindig helyes.

Nézzünk meg néhány típusátalakítást:

Turbo Pascal	Turbo C
<változó>	<változó> =
:=<típus>(<kifejezés>);	(<típus>)<kifejezés>;
var Ch : char;	**char** ch;
I:=integer(Ch);	i = (**int**) ch;

F1.4.3. Konstansok, változók tárolása, kezdetiérték-adása

A Turbo Pascal nem ad kezdőértéket a deklarált változóknak. Csak a típusos konstansok tartják meg értékeiket két alprogramhívás között. C-ben az összes globális változó a 0 kezdőértéket veszi fel, hacsak nem adunk más értéket nekik.

F1.4.3.1. Konstans típusok

A C nyelvben kétfajta konstansmegadási mód van:

```
#define <cnév> <érték>
const <típus> <cnév> = <érték>;
```

A első eset (*#define*) nagyon hasonlít a Pascal **const** definícióhoz abban, hogy az <érték> közvetlenül helyettesítődik mindenütt, ahol <cnév> megtalálható.
A második forma (**const** ...) a Turbo Pascal típusos konstansaihoz hasonlít, kivéve azt, hogy valóban nem változtatható meg a <cnév> értéke, ennek módosításakor fordítási hibát kapunk.

F1.4.3.2. Változó kezdőértéke

Turbo C-ben bármely változónak adhatunk kezdőértéket. Ez a lehetőség a Turbo Pascal típusos konstansaihoz hasonlít.

```
<típus> <vnév> = <érték>;
```

Azon változók esetén, amelyek egynél több értéket kívánnak (tömbök, struktúrák), a kezdőértéket kapcsos zárójelben, vesszővel elválasztva kell felsorolni.

```
int x = 10, y = -12;
char nev[] = "Zoli";
char kerdes = 'N';
int  valasz = 5;
char lista[2][10] = {"Elso", "Masodik"};
```

F1.4.3.3. Változók tárolása

A C többféle tárolási osztályt használ, a két legjelentősebb az **extern** és az **auto** (lokális). A globális változók (amelyek minden függvényen, beleértve a **main** függvényt is, kívül vannak a definíciók) az alapértelmezés szerint **extern** változók.
A lokális változók elvesztik az értéküket két függvényhívás között. Ha ezeket a változókat **static**-nak deklaráljuk, akkor zérus kezdőértéket kapnak (egyszer, a program végrehajtásának az elején), és az értékeik megmaradnak a két függvényhívás között is

F1.4.3.4. Dinamikus memóriafoglalás

Nézzük meg először, hogy a Turbo Pascal hogyan kezeli a *heap* (halom) területet.

```
type
    ItemType  = Integer;
    ItemPtr   = ^Integer;
var
   p : ItemPtr;
     { New, Dispose }
     New(p);                        { lefoglalja a kívánt memóriát }

     Dispose(p);                    { felszabadítja a lefoglalt
                                      memóriát }

     { Freemem és Getmem }
     Getmem(p, Sizeof(ItemType));   { meg kell adni a lefoglalandó
                                      memória nagyságát }
     Freemem(p, Sizeof(ItemType)); { meg kell adni a
                                      felszabadítandó
                                       memória nagyságát }
```


A Turbo C-ben a dinamikus memóriafoglalás a Turbo Pascal ***Getmem*** és ***Dispose*** eljárásaihoz hasonlít.

```
<típus> *<ptr>;
<ptr>  = (<típus>*) calloc(<darabszám>,<méret>);
```

vagy

```
<ptr> = (<típus>*) malloc(<teljes méret>);

free(<ptr>);
```

Példa:

```
typedef int ItemType;
ItemType  *p;

p = (ItemType*)malloc(sizeof(ItemType));
*p = 2;

free(p);
```

Minden C memóriafoglaló függvény (***calloc***, ***malloc*** és a ***realloc***) általános pointerrel (void*) tér vissza, amelyet az aktuális típusra át kell alakítani. Mind a három NULL értéket ad vissza, ha nem áll rendelkezésre a szükséges méretű memória a *heap*-en.

- A ***calloc*** függvénynek át kell adni a tételek számát és a tétel méretét (*byte*-ban), a függvény kiszámítja a blokk méretét, lefoglalja és lenullázza azt, majd visszatér a blokkra mutató pointerrel. Nagyon kényelmes ezzel a tömböknek dinamikusan helyet foglalni.
- A ***malloc*** függvénynek meg kell adni a lefoglalandó byte-ok számát.
- A ***free*** függvény pedig felszabadítja a memóriát, ahová a <*ptr*> pointer mutat.

F1.4.4. Parancssor argumentumok

Ha olyan Pascal programot írunk, amely parancssorból dolgoz fel paramétereket, akkor a *ParamCount* és a *ParamStr* függvényeket kell használni a paraméterek kezelésére.

Ha a MASOL EXE programot

```
C:\>masol sajat.txt masik.txt
```

utasítással indítjuk, akkor a *ParamCount* függvény értéke 2 lesz, és a *ParamStr* pedig a következő értékekkel tér vissza:

```
ParamStr(1)   sajat.txt
ParamStr(2)   masik.txt
```

Hasonlóan a Turbo C, követve a szabványos C konvenciókat, megengedi az *argc*, *argv* paraméterek deklarálását a **main** függvény paramétereként:

```
main(int argc, char *argv[]);
{
 /* A fõprogram törzse */
}
```

ahol az

argc - az paraméterek száma,
argv - sztring tömb, amely a paramétereket tárolja,

A példában az *argc* 3, az *argv* pedig az alábbiakat tartalmazza:

```
argv[0]    c:\MASOL.EXE
argv[1]   sajat.txt
argv[2]   masik.txt
```

F1.4.5. File I/O

A szabványos Pascal-ban kétféle file-t használhatunk:
- szöveges, amely *text*-nek van deklarálva és
- adat, amely *file of <típus>* -nak van deklarálva, típusos file.

A file nyitása, módosítása és a zárása majdnem azonos mind a két típusnál. A Turbo Pascal-nak van még egy harmadik típusú file-ja, ez ún. típus nélküli file, amely nagyon hasonlít a Turbo C bináris állományaihoz.

C file-ok gyakran byte folyammal (*stream*) kezelhetők. Megkülönböztetünk szöveges "t" (*text*) és bináris "b" (*binary*) file-t. A file típust a fopen függvényben kell jelezni.

Turbo Pascal	Turbo C
var	
I : integer;	**int** i;
X : single;	**float** X;
Ch : char;	**char** ch;
Sor : string[80];	**char** sor[81];
SRec : RT;	struct rt srec;
buf :array[1..1024] of char;	**char** buf[1024];
F1 : text;	FILE *f1;
F2 : file of RT;	FILE *f2;
F3 : file;	FILE *f3;
assign(<fvar>,<fnév>);	<fvar>=fopen(<fnév>,"r");
reset(<fvar>);	vagy <fvar>=fopen(<fnév>,"r+");
	vagy f1 =fopen(<fnév>,"r+t");
reset(<típusnélküli fvar>,	vagy f2 =fopen(<fnév>,"r+b");
<blokkméret>);	
assign(<fvar>,<fnév>);	<fvar> = fopen(<fnév>,"w");
	vagy <fvar>=fopen(<fnév>,"w+");
rewrite(<fvar>);	vagy f1 = fopen(<fnév>,"w+t);
rewrite(<típusnélküli fvar>,	vagy f2 = fopen(<fnév>,"w+b");
<blokkméret>);	
assign(<fvar>,<fnév>);	<fvar>= fopen(<fnév>,"a+");
append(<text fvar>);	vagy <fvar>=fopen(<fnév>,"a+t");
	vagy <fvar>=fopen(<fnev>,"a+b");
read(F1,Ch);	ch=getc(f1);
readln(F1,Sor);	fgets(sor,80,f1);
readln(F1,I,X);	fgets(sor,80,f1);
	sscanf(sor,"%d %f",&i,&x);
read(F2,Srec);	fread(&srec,sizeof(srec),1,f2);
BlockRead(F3,buf,Sizeof(buf),ok);	ok=fread(buf,sizeof(buf),f3);
write(F1,Ch);	fputc(ch,f1);
	vagy printf(f1,"%c",ch);
write(F1,Sor);	fputs(sor,f1);
	vagy printf(f1,"s",sor);
write(F1,I,X);	fprintf(f1,"%d %f",i,x);
writeln(F1,I,X);	fprintf(f1,"%d %f\n",i,x);
write(F2,SRec);	fwrite(&myrec,sizeof(srec),1,f2);
Seek(F2,<rec>);	fseek(f2,<rec>*sizeof(rt),0);
Flush(<fvar>);	fflush(<fvar>);
Close(<fvar>);	fclose(<fvar>);
BlockWrite(F3,buf,Sizeof(buf),ok);	fwrite(buf,1,siezof(buf),f3);

Nézzünk egy példát, amelyben a parancssorral megadott *text* file tartalmát kiírjuk a képernyőre.

Turbo Pascal

```
program Kiir;
var
  F  : text;
  Ch : char;
begin
  Assign(f,ParamStr(1));
 {$I-} Reset(F); {$I+}
 if IOResult <>0 then begin
   writeln('Nem létezik:',
                 ParamStr(1));
    Halt(1);
   end;
 while not Eof(F) do begin
    Read(F,Ch);
    Write(Ch);
 end;
 Close(F);
end.
```

Turbo C

```
#include <stdio.h>
#include <stdlib.h>
main(int argc, char *argv[])
{
 FILE *f;
 int    ch;

 f =fopen(argv[1],"r");
 if (f == NULL) {
   printf("Nem létezik: %s\n",
                   argv[1]);
   exit(1);
}
while ((ch = getc(f)) != EOF)
   putchar(ch);

  fclose(f);
}
```


F2. Turbo C 2.0 include file-ok és könyvtári függvények

A Turbo C futás idejű könyvtára (*Run-time library*) több száz rutint (függvényt és makrot) tartalmaz. Ezek egy része a rendszerspecifikus műveleteket rejti szabványos kezelői felület mögé (adat be- és kivitel, file-műveletek, időkezelés, tárfoglalás, folyamatvezérlés, stb.), más részük pedig a programírás során gyakran előforduló feladatokat lát el (karakterlánc-műveletek, adatkonverziós és diagnosztikai rutinok, stb)

A lefordított könyvtári függvények kódját a C*x*.LIB, MATH*x*.LIB, EMU.LIB, FP87.LIB és GRAPHICS LIB nevű könyvtárak tartalmazzák. Az *x* helyén a Turbo C által használt memóriamodellek (*tiny, small, medium, compact, large* és *huge*) közül a megfelelő modell nevének kezdőbetűje áll (a *tiny* és a *small* modellek közös könyvtárral rendelkeznek.)

A függelék első részében áttekintjük az include állományokat (kiemelve a bennük definiált globális változókat és típusokat), majd a könyvtári rutinok funkció szerinti csoportosítása következik.

F2.1. A Turbo C 2.0 include file-jai

Az include file-ok tartalmazzák a könyvtári függvények deklarációját, a függvények által használt adattípusokat és szimbolikus állandókat, illetve a futtató rendszer globális változóit A Turbo C követi az ANSI C szabványnak az include file-ok elnevezésére és tartalmára vonatkozó ajánlását. Azonban az ANSI C szabványt kiegészíti UNIX és IBM PC specifikus részekkel (pl. grafika). Az alábbi felsorolásban csillaggal (*) jelöltük az ANSI szabványnak megfelelő fejléc állományokat.

Include file	ANSI	Tartalom
ALLOC.H		Memóriakezelő függvények (tárfoglalás, felszabadítás, stb.).
ASSERT.H	*	Az ***assert*** hibakereső makro.
BIOS.H		Az IBM-PC BIOS-rutinjainak közvetlen hívásához használható függvények.
CONIO.H		A konzol (BIOS szintű) I/O függvények.
CTYPE.H	*	A karaktert osztályozó és karaktert átalakító makrok.

Include file	ANSI	Tartalom
DIR.H		A könyvtárkezeléshez tartozó függvények és makrok.
DOS.H		Az MS-DOS- és a 8086-specifikus hívásokhoz szükséges deklarációk és definíciók.
ERRNO.H	*	A szabványos hibakódokhoz tartozó szimbolikus konstansok.
FCNTL.H		Az ***open*** könyvtári függvény használatához szükséges szimbolikus állandók.
FLOAT.H	*	A lebegőpontos függvények paraméterei.
GRAPHICS.H		A Borland grafikus felület használatához szükséges konstansok és függvények.
IO.H		Struktúrák és deklarációk az alacsonyszintű I/O függvények használatához.
LIMITS.H	*	Az egész típusú mennyiségek értékhatárai.
MATH.H	*	Matematikai függvények, a ***HUGE_VAL*** makro, és a ***matherr*** függvény által használt struktúra.
MEM.H		Memóriakezelő függvények. (Többsége a ***STRING.H***-ban is megtalálható.)
PROCESS.H		A ***spawn...*** és az ***exec..*** függvények.
SETJMP.H	*	A ***longjmp*** és a ***setjmp*** függvények és az ezek által használt ***jmp_buf*** típus.
SHARE.H		Az osztott file-kezeléshez tartozó függvények paraméterezéséhez használt konstansok.
SIGNAL.H	*	Konstansok és típusok a ***signal*** és a ***raise*** függvények használatához.
STDARG.H	*	Makrok a függvények változó hosszúságú argumentum-listájának feldolgozására.
STDDEF.H	*	Különböző rendeltetésű adattípusok és makrok
STDIO.H	*	A szabványos I/O műveletek elvégzéséhez szükséges típusok, makrok és függvények.
STDLIB.H	*	Általános célú függvények (rendezés, keresés, konverzió, stb.)

Include file	ANSI	Tartalom
STRING.H	*	Sztring- és memóriakezelő függvények.
SYS\STAT.H		A file-ok megnyitásakor, illetve létrehozásakor használt szimbolikus konstansok.
SYS\TIMEB.H		Az ***ftime*** függvény és a hozzá tartozó adatstruktúra.
SYS\TYPES.H		A ***time_t*** típus.
TIME.H	*	Időátalakító függvények és a hozzájuk tartozó adatstruktúrák.
VALUES.H		Fontos (gépfüggő) konstansok. Kompatibilis a UNIX V. rendszerrel.

F2.2. Globális változók és szabványos típusok

A C Run-Time könyvtár tartalmaz néhány olyan változó- és típusdefiníciót, amelyeket a könyvtári rutinok használnak Ezen változók és típusok elérhetők a megfelelő include file-ok becsatolásával

F2.2.1. Globális változók

extern int *_argc;*	dos.h

A parancssor argumentumainak számát tartalmazza

extern char *_argv[];	dos.h

A parancssor argumentumait tartalmazó sztring tömb.

extern int *daylight;	time.h
extern long *timezone;	time.h
extern char *tzname[2];	time.h

A fenti változókat az idő- és dátum függvények többsége használja a helyi időre történő igazításhoz.

extern int *directvideo;	conio.h

A ***directvideo*** változó értéke a konzolra történő kiírás módját határozza meg. (0-BIOS hívással, 1-a képernyő memória közvetlen írásával)

extern int *_8087;* — dos.h

A változó értéke megmutatja, hogy a gépben van-e matematikai társprocesszor (1-8087, 2-80287, 3-80387) vagy nincs (0).

extern int *_doserrno*, — dos.h
extern int *errno*, — errno.h
extern char **sys_errlist*[]; — stdlib.h
extern int *sys_nerr*, — stdlib.h

Ezen változókat elsősorban a ***perror*** függvény használja a hibainformációk kiírására

extern int *_fmode;* — fcntl.h

Az ***_fmode*** változó a file-ok feldolgozási módját határozza meg. Az alapértelmezés (0) szerinti mód a szöveges mód

extern unsigned *_heaplen*; — dos.h

A közeli (***near***) halom méretét tartalmazza. (A 0 érték esetén maximális lesz a halomterület mérete.)

extern unsigned *_stklen*; — dos.h

A stack méretét tartalmazza. (A stack minimális mérete 128 szó, míg az alapmérete 4 KByte).

unsigned char *_osmajor*; — dos.h
unsigned char *_osminor*; — dos.h
unsigned *_version*; — dos.h

A használt DOS operációs rendszer verziószámát adják meg.

extern char **environ*[]; — dos.h

Az ***environ*** mutatótömb lehetővé teszi a DOS környezeti információk közvetlen elérését.

extern unsigned int *_psp*; — dos.h

A program PSP (program szegmens előtag) szegmenscímét tartalmazza.

F2.2.2. Szabványos típusok

BYTEREGS — dos.h
WORDREGS

A processzor byte-os és szavas regisztereit lefedő adatstruktúrák.

clock_t — time.h

Típus az idő értékek tárolására.

complex — math.h

Komplex számok tárolására használható struktúra

country — dos.h

Struktúra az országfüggő információk tárolására.

devhdr — dos.h

Struktúra az MS-DOS eszközvezérlők fejlécének tárolására.

dfree — dos.h

Struktúra a lemezinformációk tárolására.

div_t — stdlib.h
ldiv_t

Struktúra a ***div*** és ***ldiv*** függvények által visszaadott értékek tárolására.

date — dos.h

Struktúra az aktuális dátum tárolására.

DOSERROR — dos.h

Struktúra az 59H MS-DOS rendszerhívás által visszaadott értékek tárolására.

exception — math.h

Struktúra a matematikai rutinok hibainformációinak tárolására.

fatinfo — dos.h

Struktúra a FAT (file helyfoglalási tábla) információinak tárolására

fcb — dos.h

Struktúra az MS-DOS FCB (file-vezérlő blokk) tárolására.

xfcb — dos.h

Struktúra a bővített MS-DOS FCB (file-vezérlő blokk) tárolására.

ffblk — dos.h

Struktúra a FIB (file információs blokk) tárolására.

FILE — stdio.h

A stream I/O műveletek során az állomány azonosítására szolgáló struktúra.

fpos_t — stdio.h

Struktúra a file-on belüli pozíció tárolására

jmp_buf[] — setjmp.h

Puffer a program adott pontjához tartozó *task*-állapot elmentésére.

_mexcep — math.h

Enum típus a matematikai hibákhoz.

ptrdiff_t — alloc.h, mem.h

Mutatók különbségének tárolására szolgáló adattípus.

REGPACK — dos.h

Struktúra az ***intr*** függvény használatához.

REGS — dos.h

Unió típus a különböző megszakításhívások paraméterezéséhez.

size_t — stddef.h, ...

A memória objektumok méretének és az ismétlések számának megadásához használt típus.

sig_automatic_t — signal.h

A szignálok kezeléséhez használt szabványos típus

SREGS — dos.h

Struktúra a szegmensregiszterek (ES, CS, SS és DS) értékének tárolására.

stat — sys\stat.h

Struktúra a file-állapot információk tárolására.

text_info — conio.h

Struktúra a szöveges ablak információinak tárolására

time_t — time.h

A ***time_t*** típus az időt reprezentálja az ***mktime***, ***time*** függvények számára.

timeb — sys\timeb.h

Az ***ftime*** függvény hívásánál használt struktúra.

tm — time.h

Struktúra az ***asctime***, ***gmtime*** és ***localtime*** függvények időinformációjának tárolására.

va_list — time.h

A változó hosszúságú argumentumlista feldolgozásához használt mutatótípus.

F2.2.3. A BGI könyvtár globális változói és típusai

Az alábbi definíciókat a GRAPHICS.H include file tartalmazza.

enum ***graphics_errors***
A grafikus hibakódokat tartalmazó típus.

enum ***graphics_drivers***
A BGI vezérlőket tartalmazó típus

enum ***graphics_modes***
A beállítható grafikus módokat tartalmazó típus.

enum ***COLORS***
A 16 alapszínt tartalmazó típus.

enum ***CGA_COLORS***
A CGA vezérlő palettaszíneit tartalmazó típus.

enum ***EGA_COLORS***
Az EGA vezérlő színkódjait tartalmazó típus.

enum ***line_styles***
A vonalfajtákat tartalmazó típus.

enum ***line_widths***
A vonalvastagságokat tartalmazó típus.

enum ***font_names***
A BGI karakterkészletek neveit tartalmazó típus.

enum ***fill_patterns***
A kitöltési mintákat tartalmazó típus.

enum ***putimage_ops***
A képekkel végezhető műveleteket tartalmazó típus.

enum ***text_just***
A szövegkiigazítási lehetőségeket tartalmazó típus.

struct ***palettetype***
A paletta elérésére szolgáló struktúra.

struct ***linesettingstype***
A vonaltípus beállításához használható struktúra.

struct ***textsettingstype***
A szöveg megjelenésének beállításához használható típus.

struct ***fillsettingstype***
A kifestési mód kiválasztásához használható struktúra.

struct ***pointtype***
A képpontkoordinátákat tartalmazó típus.

struct ***viewporttype***
A *viewport* beállításához használható típus.

struct ***arccoordstype***
Az ívek rajzolásánál a jellemző pontokat tartalmazó típus.

F2.3. A könyvtári függvények csoportosítása

F2.3.1. Karakterek osztályozása és konverziója

ctype h

Az *is...* makrok segítségével az ASCII karakterek számjegyekre, vezérlő karakterekre, írásjelekre, kis- és nagybetűkre, stb. való osztályozása végezhető el A *.to...* függvények és makrok a karakterek egyszerű átalakítására szolgálnak

Függvény	Leírás	Turbo C	ANSI C	UNIX C
isalnum	Alfanumerikus karakter tesztelése	✓	✓	✓
isalpha	Alfabetikus karakter tesztelése	✓	✓	✓
isascii	ASCII karakter tesztelése	✓	✓	✓
iscntrl	Vezérlő karakter tesztelése.	✓	✓	✓
isdigit	Decimális számjegy tesztelése.	✓	✓	✓
isgraph	Kinyomtatható karakter tesztelése (de nem space).	✓	✓	✓
islower	Angol kisbetű tesztelése	✓	✓	✓
isprint	Kinyomtatható karakter tesztelése	✓	✓	✓
ispunc	Irásjel karakter tesztelése	✓	✓	✓
isspace	Szóköz, tabulátor, soremelés vagy lapdobás karakter tesztelése.	✓	✓	✓
isupper	Angol nagybetű tesztelése.	✓	✓	✓
isxdigit	Hexadecimális számjegy tesztelése.	✓	✓	✓
toascii	Karakter konvertálás ASCII karakterré.	✓		✓
tolower	Karakter tesztelése és konvertálása kisbetűvé, ha az nagybetű	✓	✓	✓
_tolower	Karakter kisbetűvé alakítása vizsgálat nélkül	✓		✓
toupper	Karakter tesztelése és konvertálása nagybetűvé, ha az kisbetű.	✓	✓	✓
_toupper	Karakter nagybetűvé alakítása vizsgálat nélkül.	✓		✓

F2.3.2. Adatkonverziók

stdlib.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
atof	Sztring konvertálása lebegőpontos (**double**) értékké	✓	✓	✓
atoi	Sztring konvertálása **int** típusúvá	✓	✓	✓
atol	Sztring konvertálása **long** típusúvá.	✓	✓	✓
ecvt	**double** konvertálása sztringgé, a tizedespont és az előjel elhelyezése nélkül	✓		✓
fcvt	**double** konvertálása sztringgé, a tizedespont és az előjel elhelyezése nélkül.	✓		✓
gcvt	**double** sztringgé konvertálása tizedestört alakban	✓		✓
itoa	**int** konvertálása sztringgé	✓		
ltoa	**long** konvertálása sztringgé.	✓		
strtod	Sztring konvertálása **double** típusúvá.	✓	✓	✓
strtol	Sztring konvertálása **long int** típusú egésszé.	✓	✓	
strtoul	Sztring konvertálása **unsigned long int** típusú egésszé	✓	✓	
ultoa	**unsigned long** konvertálása sztringgé.	✓		

F2.3.3. Könyvtárak kezelése

dir h

Az alábbi függvények segítségével a könyvtárakat és az elérési útvonalakat kezelhetjük

Függvény	Leírás	Turbo C	ANSI C	UNIX C
chdir	Aktuális alkönyvtár beállítása	✓		✓
findfirst	Valamely file első előfordulásának keresése	✓		
findnext	Valamely file következő előfordulásának keresése.	✓		
fnmerge	Az elérési útvonal felépítése komponensekből.	✓		
fnsplit	Az elérési útvonal szétszedése komponensekre	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
getcurdir	A megadott meghajtó aktuális alkönyvtárának lekérdezése.	✓		
getcwd	Az aktuális alkönyvtár lekérdezése.	✓		
getdisk	Az aktuális meghajtó lekérdezése.	✓		
mkdir	Új alkönyvtár létrehozása.	✓		✓
mktemp	Egyedi file-név előállítása	✓		✓
rmdir	Alkönyvtár törlése.	✓		✓
searchpath	File keresése a rendszer PATH felhasználásával.	✓		
setdisk	Az aktuális meghajtó beállítása.	✓		

F2.3.4. Input és output rutinok

A C nyelvben nincsenek előre definiált file-struktúrák, minden adatot byte-ok sorozataként kezelünk. Az I/O függvényeknek a következő formái lehetségesek:

- *Stream* (folyam) I/O
  Ezen függvények a file-okat, mint karakterek folyamát tekintik. Amikor egy file a stream-függvények használatával kerül megnyitásra, a nyitott állományt egy ***FILE*** struktúrával, illetve a ***FILE*** struktúrára mutató pointerrel azonosítja a rendszer. A stream-függvények pufferelt, formázott vagy formázatlan adatforgalmat képesek lebonyolítani A Turbo C programban öt szabványos stream-re hivatkozhatunk, amelyek megnyitását a futtató rendszer automatikusan elvégzi:

Stream	Leíró	Jelentés
stdin	*0*	a szabványos input stream
stdout	*1*	a szabványos output stream
stderr	*2*	a szabványos hiba stream
stdaux	*3*	a szabványos másodlagos I/O stream
stdprn	*4*	a szabványos nyomtató stream

- Alacsonyszintű I/O
  Az alacsonyszintű I/O függvények lehetővé teszik a file-ok és a perifériális eszközök operációs rendszer szintjén történő elérését. A file-ok azonosítása ezen a szinten a leíróval (file-szám, *handle*) történik.

– Konzol és port I/O

A konzol függvények az operációs rendszert megkerülve közvetlenül vagy a ROM BIOS felhasználásával végzik el az I/O műveleteket. Ugyancsak itt találhatók függvények, melyek segítségével mód nyílik a szöveges képernyő hatékony kezelésére.

F2.3.4.1. Stream rutinok

stdio.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
clearerr	A hibajelző törlése	✓	✓	✓
fclose	A stream lezárása.	✓	✓	✓
fcloseall	Minden megnyitott stream lezárása	✓	✓	✓
fdopen	Stream megnyitása file-leíró felhasználásával.	✓		✓
feof	A file-vége esemény tesztelése.	✓	✓	✓
ferror	Hiba tesztelése a stream-en	✓	✓	✓
fflush	Az output stream puffertartalmának kiírása a file-ba - az input stream pufferének törlése.	✓	✓	✓
fgetc	Karakter olvasása a stream-ből	✓	✓	✓
fgetchar	Karakter olvasása az ***stdin***-ről	✓		✓
fetpos	A file-pozíció lekérdezése	✓	✓	
fgets	Sztring olvasása a stream-ből	✓	✓	✓
fileno	A stream-hez tartozó file-leíró lekérdezése	✓		✓
flushall	Mint az ***fflush,*** de az összes megnyitott stream-re vonatkoztatva	✓		✓
fopen	A stream megnyitása.	✓	✓	✓
fprintf	Formázott adatok írása a stream-be	✓	✓	✓
fputc	Karakter írása a stream-be.	✓	✓	✓
fputchar	Karakter írása az ***stdout***-ra	✓		✓
fputs	Sztring írása a stream-be.	✓	✓	✓
fread	Formázatlan adatok olvasása stream-ből.	✓	✓	✓

Függvény	Leírás	Turbo C	ANSI C	UNIX C
freopen	Új file hozzárendelése a már megnyitott stream-hez.	✓	✓	✓
fscanf	Formázott adatok olvasása stream-ből.	✓	✓	✓
fseek	A file-mutató beállítása adott pozícióra (relatíven a file elejéhez, aktuális pozíciójához vagy a végéhez).	✓	✓	✓
fsetpos	A file-mutató beállítása adott pozícióra.	✓	✓	
ftell	Az aktuális file-pozíció lekérdezése	✓	✓	✓
fwrite	Formázatlan adatelemek (memóriablokk) írása stream-be.	✓	✓	✓
getc	Karakter olvasása stream-ből (makro).	✓	✓	✓
getchar	Karakter olvasása az ***stdin***-ről (makro).	✓	✓	✓
gets	Szövegsor olvasása az ***stdin***-ről	✓	✓	✓
getw	Bináris ***int*** érték olvasása stream-ből.	✓		✓
perror	Hibaüzenet kiírása a ***stderr***-re.	✓	✓	✓
printf	Formázott adatok írása az ***stdout***-ra.	✓	✓	✓
putc	Karakter írása stream-be (makro)	✓	✓	✓
putchar	Karakter írása az ***stdout***-ra (makro).	✓	✓	✓
puts	Egy sor írása az ***stdout***-ra	✓	✓	✓
putw	Bináris ***int*** érték írása stream-be.	✓		✓
rewind	A file-mutató ráállítása a stream kezdetére.	✓	✓	✓
scanf	Formázott adatok olvasása az ***stdin***-ről.	✓	✓	✓
setbuf	A stream pufferhasználatának egyszerű vezérlése.	✓	✓	✓
setvbuf	A stream pufferhasználatának finombeállítása.	✓	✓	✓
sprintf	Formázott adatok sztring-be írása	✓	✓	✓
sscanf	Formázott adatok olvasása sztring-ből	✓	✓	✓
tmpfile	Ideiglenes file létrehozása.	✓	✓	✓
tmpnam	Ideiglenes file-név előállítása.	✓	✓	✓
ungetc	Egy karakter elhelyezése az input pufferban.	✓	✓	✓
vfscanf	Formázott adatok olvasása stream-ről, ***va_list*** típusú argumentumot használva.	✓		✓

Függvény	Leírás	Turbo C	ANSI C	UNIX C
vfprintf	Formázott adatok stream-be írása, *va_list* típusú argumentumot használva.	✓	✓	✓
vprintf	Formázott adatok *stdout*-ra írása, *va_list* típusú argumentumot használva.	✓	✓	✓
vscanf	Formázott adatok olvasása *stdin*-ről, *va_list* típusú argumentumot használva.	✓		✓
vsprintf	Formázott adatok sztring-be írása, *va_list* típusú argumentumot használva.	✓	✓	✓
vsscanf	Formázott adatok olvasása sztring-ből, *va_list* típusú argumentumot használva.	✓		✓

F2.3.4.2. Alacsonyszintű I/O rutinok

io.h, fcntl.h, sys\types.h, sys\stat.h, stdio.h, dir.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
access	Adott file elérhetőségének ellenőrzése	✓		✓
chmod	Adott file elérhetőségének beállítása.	✓		✓
_chmod	A file-attribútumok beállítása és cseréje.	✓		
chsize	A file-méret megváltoztatása.	✓		
close	Adott leírójú file lezárása.	✓		✓
_close	Adott leírójú file lezárása.	✓		
creat	Új file létrehozása / létező file felülírása, a file elérhetőségének megadásával.	✓		✓
_creat	Új file létrehozása / létező file felülírása, a file-attribútumok beállításával.	✓		
creatnew	Új file létrehozása, a file-attribútumok beállításával.	✓		
creattemp	Egyedi file létrehozása, a file-attribútumok beállításával.	✓		
dup	Egy második leíró hozzárendelése a file-hoz.	✓		✓
dup2	Érvényes leíró lecserélése megadott leíróra.	✓		✓
eof	A file-vége esemény tesztelése.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
filelength	A file-hossz lekérdezése.	✓		
fstat	Megnyitott file állapotát tartalmazó információk elérése.	✓		✓
getftime	File létrehozási idejének tesztelése.	✓		
ioctl	Az I/O egység vezérlése (0x44 DOS-funkció).	✓		✓
isatty	Valamely perifériális egység karakteres voltának ellenőrzése (*character device*).	✓		✓
lock	Adott file valamely részének lezárása.	✓		
lseek	A file-mutató beállítása adott pozícióra (relatíven a file elejéhez, aktuális pozíciójához vagy a végéhez).	✓		✓
mktemp	Egyedi file-név létrehozása.	✓		✓
open	File megnyitás, az elérési mód megadásával.	✓		✓
_open	File megnyitása, a DOS-attribútumok megadásával	✓		
read	Memóriablokk olvasása a file-ból (max. méret 65534).	✓		✓
_read	Memóriablokk olvasása a file-ból (max. méret 65534).	✓		
remove	File törlése.	✓	✓	✓
rename	File átnevezése.	✓	✓	
setftime	File létrehozási idejének beállítása.	✓		
setmode	Nyitott file elérési módjának beállítása	✓		✓
sopen	File megnyitása osztott eléréshez.	✓		✓
stat	Nevével megadott file állapot-információinak lekérdezése	✓		✓
tell	A file-mutató aktuális pozíciójának lekérdezése.	✓		✓
unlink	File törlése.	✓		✓
unlock	A lezárt file-rész feloldása.	✓		
write	Memóriablokk file-ba írása (max. méret 65534).	✓		✓
_write	Memóriablokk file-ba írása (max. méret 65534).	✓		

F2.3.4.3. Konzol és port I/O

conio.h, dos.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
cgets	Sztring olvasása a billentyűzetről	✓		
cprinf	Formázott adat írása a képernyőre.	✓		
cputs	Sztring írása a képernyőre.	✓		
cscanf	Formázott adat olvasása a billentyűzetről.	✓		✓
getch	Karakter olvasása a billentyűzetről.	✓		
getche	Karakter olvasása a billentyűzetről, visszaírással.	✓		
getpass	Jelszó bekérése	✓		✓
inportb	Byte beolvasása adott I/O portról.	✓		
inport	Két byte-os szó beolvasása adott I/O portról.	✓		
kbhit	Billentyűnyomás ellenőrzése a konzolon.	✓		
outportb	Byte kiírása adott I/O portra.	✓		
outport	Két byte-os szó kiírása adott I/O portra.	✓		
putch	Karakter kiírása a képernyőre	✓		
ungetch	Az utoljára olvasott karakter visszaírása a billentyűzet pufferba.	✓		✓
clreol	Törlés a képernyőn a kurzor pozíciójától a sor végéig.	✓		
clrscr	Szöveges ablak törlése.	✓		
delline	Sor törlése szöveges ablakban.	✓		
gettext	Szövegblokk másolása képernyőről a memóriába.	✓		
gettextinfo	Szöveges ablak információinak lekérdezése.	✓		
gotoxy	Pozícionálás a szöveges ablakban.	✓		
highvideo	Magas intenzitású karaterszín beállítása.	✓		
insline	Sor beszúrása a kurzort tartalmazó sorba.	✓		
lowvideo	Alacsony intenzitású karakterszín beállítása.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
movetext	Szövegblokk mozgatása a képernyőn.	✓		
normvideo	Normális intenzitású karakterszín beállítása.	✓		
puttext	Szövegblokk másolása memóriából képernyőre.	✓		
textattr	Az attribútum beállítása a szöveges ablakon belül.	✓		
textbackground	A háttérszín beállítása	✓		
textcolor	Az előtérszín beállítása	✓		
textmode	Szöveges üzemmód beállítása	✓		
wherex	Kurzor vízszintes pozíciójának lekérdezése	✓		
wherey	Kurzor függőleges pozíciójának lekérdezése	✓		
window	Szöveges ablak definiálása.	✓		

F2.3.5. Matematikai rutinok

math.h, float.h

A C nyelvre jellemző sokszínűség a matematikai függvények csoportját is áthatja. A szokásos függvényeken (abs, sin, cos, sqrt) túlmenően egy sor különleges függvényt is megtalálhatunk (hypot, poly, stb).

Függvény	Leírás	Turbo C	ANSI C	UNIX C
abs	**int** típusú kifejezés abszolútértéke (y=|x|)	✓	✓	✓
acos	Arcus cosinus függvény (0-π, y=arc cos x)	✓	✓	✓
asin	Arcus sinus függvény (-π/2 - π/2, y=arc sin x)	✓	✓	✓
atan	Arcus tangens függvény (-π/2 - π/2, y=arc tg x)	✓	✓	✓
atan2	Arcus tangens függvény (-π - π, y=arc tg a/b)	✓	✓	✓
cabs	***complex*** típusú kifejezés abszolútértéke. (y=|x|)	✓	✓	✓
ceil	A nagyobb egészek közül a legkisebb megkeresése.	✓	✓	✓
_clear87	Lebegőpontos állapotszó lekérdezése és törlése.	✓		
_control87	A lebegőpontos vezérlőszó lekérdezése és az új vezérlőszó betöltése.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
cos	Cosinusfüggvény (y=cos x)	✓	✓	✓
cosh	Cosinus hiperbolicus függvény. (y=ch x)	✓	✓	✓
div	**int** számok osztása a hányados és a maradék visszaadásával.	✓	✓	
exp	Exponenciális függvény. ($y=e^x$)	✓	✓	✓
fabs	**double** típusú kifejezés abszolútértéke. (y=|x|)	✓	✓	✓
floor	A kisebb egészek közül a legnagyobb megkeresése. (Egészrész függvény. y=[x])	✓	✓	✓
fmod	Lebegőpontos maradék számítása.	✓	✓	✓
_fpreset	A lebegőpontos rutincsomag újrainicializálása.	✓		
frexp	A **double** típusú argumentum felbontása mantisszára és 2 hatványára.	✓	✓	✓
hypot	Derékszögű háromszög befogóihoz tartozó átfogó kiszámítása.	✓		✓
labs	**long** típusú kifejezés abszolútértéke. (y=|x|).	✓	✓	✓
ldexp	Mantisszából és 2 hatványkitevőjéből egy lebegőpontos szám előállítása	✓	✓	✓
ldiv	**long** típusú számok osztása a hányados és a maradék visszaadásával	✓	✓	
log	Természetes alapú logaritmus függvény (y=ln x)	✓	✓	✓
log10	Tízes alapú logaritmus függvény (y=lg x)	✓	✓	✓
matherr	Matematikai hibák programozó által definiált kezelése	✓		✓
modf	Lebegőpontos szám egész- és törtrészre bontása.	✓	✓	✓
poly	Polinom értékének számítása.	✓		✓
pow	Hatványozás. ($y=a^x$)	✓	✓	✓
pow10	10 hatványainak számítása. ($y=10^x$)	✓		✓
sin	Sinusfüggvény. (y=sin x)	✓	✓	✓
sinh	Sinus hiperbolicus függvény. (y=sh x)	✓	✓	✓
sqrt	Négyzetgyök függvény. ($y=\sqrt{x}$)	✓	✓	✓

Függvény	Leírás	Turbo C	ANSI C	UNIX C
_status87	A lebegőpontos állapotszó lekérdezése.	✓		
tan	Tangensfüggvény. (y=tg x)	✓	✓	✓
tanh	Tangens hiperbolicus függvény (y=th x)	✓	✓	✓

F2.3.6. Memóriakezelés

alloc h, stdlib.h

A dinamikus tárkezelő függvények segítségével memóriablokkokat foglalhatunk le (*malloc, calloc*) és szabadíthatunk fel (*free*). Lehetőség van azonban a lefoglalt terület átméretezésére is (*realloc*).

Függvény	Leírás	Turbo C	ANSI C	UNIX C
brk	Az adatszegmens helyfoglalásának megváltoztatása	✓		✓
calloc	Tárterület lefoglalása (tömb számára) a terület 0 értékű byte-okkal történő feltöltése mellett.	✓	✓	✓
coreleft	Az utolsó szabad, összefüggő memóriablokk méretének lekérdezése.	✓		
farcalloc	Tárterület lefoglalása a távoli halomterületen a terület 0 értékű byte-okkal történő feltöltése mellett.	✓		
farcoreleft	Az utolsó szabad, összefüggő memóriablokk méretének lekérdezése a távoli halomterületen	✓		
farfree	A *farcalloc, farmalloc* és a *farrealloc* függvények segítségével lefoglalt blokk felszabadítása a távoli halomterületen.	✓		
farmalloc	Memóriablokk foglalása a távoli halomterületen	✓		
farrealloc	Memóriablokk újrafoglalása a távoli halomterületen	✓		
free	A *calloc*, *malloc* és a *realloc* függvények segítségével lefoglalt blokk felszabadítása	✓	✓	✓
malloc	Memóriablokk foglalása.	✓	✓	✓
realloc	Memóriablokk újrafoglalása.	✓	✓	✓
sbrk	A break-érték újraállítása.	✓		✓

F2.3.7. Pufferek használata

mem.h, string.h

A pufferkezelő függvények a memóriának adott címem kezdődő, adott hosszúságú részein fejtik ki hatásukat. Lehetőség van memóriablokkok másolására, összehasonlítására, adott byte keresésére a blokkban, illetve a blokk feltöltésére.

Függvény	Leírás	Turbo C	ANSI C	UNIX C
memccpy	Karakterek másolása egyik pufferből a másikba adott karakter eléréséig.	✓		✓
memchr	Egy mutatót ad vissza, amely egy adott karakter első előfordulását jelzi a pufferben.	✓	✓	✓
memcmp	Két pufferen belül adott számú karakter összehasonlítása.	✓	✓	✓
memcpy	Adott számú karakter másolása egyik pufferből a másikba.	✓	✓	✓
memicmp	Mint a *memcmp*, de a kis- és nagybetűk között nem tesz különbséget.	✓		✓
memmove	Adott számú karakter mozgatása egyik pufferből a másikba. (Egymást átfedő pufferek esetén is jól működik.)	✓	✓	✓
memset	A pufferben adott számú byte feltöltése adott karakterrel.	✓	✓	✓
movedata	Karakterek másolása különböző szegmensekben elhelyezkedő pufferek között	✓		
movmem	Adott számú karakter másolása egyik pufferből a másikba (Egymást átfedő pufferek esetén is jól működik.)	✓		
setmem	Puffer feltöltése adott hosszon, adott karakterrel.	✓		

F2.3.8. Sztringkezelés

string.h

A 0 értékű byte-tal határolt memóriaterületeket (sztringeket) kezelő függvények tartoznak ebbe a csoportba. A pufferekhez definiált műveleteken túlmenően, speciális műveletek (pl. "tokenizálás") is gazdagítják a választékot.

Függvény	Leírás	Turbo C	ANSI C	UNIX C
stpcpy	Sztring bemásolása egy másikba. A függvény az eredmény sztring végére mutató pointert ad vissza.	✓		✓
strcat	Sztringek összekapcsolása	✓	✓	✓
strchr	Karakter első előfordulásának keresése sztringben.	✓	✓	✓
strcmp	Két sztring összehasonlítása.	✓	✓	✓
strcmpi	Két sztring összehasonlítása a kisbetűk és a nagybetűk megkülönböztetése nélkül	✓		
strcpy	Sztring átmásolása egy másikba. A függvény az eredmény sztring elejére mutató pointert ad vissza .	✓	✓	✓
strcspn	Egy karakterkészlet sztring karaktereinek keresése másik sztringben.	✓	✓	✓
strdup	Sztring duplázása helyfoglalással	✓		✓
strerror	Adott rendszerüzenet szövegére mutató pointer lekérdezése.	✓	✓	
_strerror	Felhasználói hibüzenet létrehozása.	✓		
stricmp	Ugyanaz, mint az *strcmpi* függvény.	✓		
strlen	A sztring hosszának lekérdezése	✓	✓	✓
strlwr	Sztring átalakítása kisbetűssé.	✓		
strncat	Valamely sztring adott számú karakterének hozzáfűzése egy másik sztringhez	✓	✓	✓
strncmp	Két sztring adott számú karakterének összehasonlítása.	✓	✓	✓
strncmpi	Mint az *strncmp* függvény, a kis- és a nagybetűk megkülönböztetése nélkül.	✓	✓	✓
strncpy	Valamely sztringből adott számú karakter átmásolása egy másik sztringbe.	✓	✓	✓
strnicmp	Ugyanaz, mint az *strncmpi* függvény.	✓		
strnset	Valamely sztring adott számú karakterének kicserélése adott karakterre.	✓		
strpbrk	Valamely sztring karaktereinek keresése egy másik sztringben.	✓	✓	✓

Függvény	Leírás	Turbo C	ANSI C	UNIX C
strrchr	Adott karakter utolsó előfordulásának keresése sztringben.	✓	✓	✓
strrev	A sztring megfordítása.	✓		
strset	Sztring feltöltése adott karakterrel.	✓		
strspn	Valamely sztring első olyan részsztringjének megkeresése, amelynek karakterei egy másik sztringben adottak.	✓	✓	✓
strstr	Valamely sztring első előfordulásának megkeresése egy másik sztringben.	✓	✓	✓
strtok	Valamely sztring részekre bontása. Az elválasztó karakterek egy másik sztringben adottak.	✓	✓	✓
strupr	Sztring átalakítása nagybetűssé.	✓		

F2.3.9. Folyamatvezérlés

process.h, signal.h, stdlib.h, setjmp.h, assert.h

A C nyelv lehetővé teszi a futó programból újabb folyamatok (*process*) indítását és megállítását. Jelzések (*signal*) segítségével programunkat bizonyos események kezelésére is felkészíthetjük. A ***setjmp*** és a ***longjump*** függvény segítségével a program tetszőleges pontjára adhatjuk át a vezérlést.

Függvény	Leírás	Turbo C	ANSI C	UNIX C
abort	A program futásának megszakítása hibajelzéssel	✓	✓	✓
assert	Hibaüzenet kiírása után, az ***abort*** függvény végrehajtása, amennyiben az ***assert*** makro argumentuma nulla értékű.	✓	✓	✓
atexit	Kilépési (*exit*) függvények regisztrálása.	✓	✓	
execl	Program végrehajtása argumentumlista átadásával.	✓		
execle	Program végrehajtása argumentumlista átadásával és környezet megadásával.	✓		
execlp	Program végrehajtása a PATH változó és az argumentumlista használatával.	✓		
execlpe	Program végrehajtása a PATH változó, környezetmegadás és argumentumlista használatával.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
execv	Program végrehajtása argumentumtömb átadással.	✓		
execve	Program végrehajtása argumentumtömb átadással és környezet megadással.	✓		
execvp	Program végrehajtása a PATH változó és az argumentumtömb használatával.	✓		
execvpe	Program végrehajtása a PATH változó, környezetmegadás és argumentumtömb használatával.	✓		
exit	Kilépés a programból, a pufferek kiürítése, a nyitott fileok lezárása és a kilépési függvények meghívása után.	✓	✓	✓
_exit	Kilépés a programból, a pufferek kiürítése, a nyitott fileok lezárása és a kilépési függvények meghívása nélkül.	✓		✓
getpid	A folyamat azonosítójának (ID) lekérdezése. (DOS alatt ez a PSP szegmenscíme - makro.)	✓	✓	✓
longjmp	A task-állapot visszaállítása.	✓	✓	✓
raise	Adott jelzés (szignál) kiváltása.	✓	✓	✓
setjmp	A task-állapot elmentése.	✓	✓	✓
signal	A szignálkezelő függvény hozzákapcsolása adott jelzéshez.	✓	✓	
spawnl	Program végrehajtása az argumentumlista átadásával és a futtatási mód beállításával.	✓		
spawnle	Program végrehajtása argumentumlista átadással és környezetmegadással.	✓		
spawnlp	Program végrehajtása a PATH változó és az argumentumlista használatával.	✓		
spawnlpe	Program végrehajtása a PATH változó, környezetmegadás és argumentumlista használatával.	✓		
spawnv	Program végrehajtása argumentumtömb átadással	✓		
spawnve	Program végrehajtása argumentumtömb átadással és környezet megadással.	✓		
spawnvp	Program végrehajtása a PATH változó és az argumentumtömb használatával.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
spawnvpe	Program végrehajtása a PATH változó, környezetmegadás és argumentumtömb használatával	✓		
system	Operációs rendszer parancsok végrehajtása.	✓	✓	✓

F2.3.10. Időhasználat

time.h, sys\timeb.h, sys\types.h

A csoport függvényeinek felhasználásával szabványos idő- és dátumkezelést, illetve időmérést valósíthatunk meg a C nyelvű programunkban

Függvény	Leírás	Turbo C	ANSI C	UNIX C
asctime	Az idő konvertálása ***struct tm*** típusú struktúrából sztringgé.	✓	✓	✓
clock	A program által felhasznált CPU idő lekérdezése	✓	✓	
ctime	Az idő konvertálása ***time_t*** formátumból sztringgé.	✓	✓	✓
difftime	Két időpont közötti időkülönbség kiszámítása (mp)	✓	✓	✓
ftime	Az aktuális rendszeridő kiolvasása **struct *timeb*** típusú struktúrába.	✓		✓
gmtime	A Greenwich-i idő átalakítása ***time_t*** formátumból ***struct tm*** struktúra formátumúvá	✓	✓	✓
localtime	Az idő átalakítása ***time_t*** formátumból ***struct tm*** típusú struktúra formátumúvá, a helyi időnek megfelelő korrekciók elvégzésével	✓	✓	✓
stime	A rendszeridő és -dátum beállítása.	✓		✓
time	A rendszeridő lekérdezése ***time_t*** alakban (A függvény az *1970 január 1 00 00 00 GMT* óta eltelt időt adja meg másodpercben.)	✓	✓	✓
tzset	A ***TZ*** környezeti változóból az időzónaérték áttöltése a globális időváltozókba	✓		✓

F2.3.11. Változó hosszúságú argumentumlista kezelése (makrok)

stdarg.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
va_arg	Az aktuális argumentum lekérése a listából.	✓	✓	✓
va_end	Az argumentumlista-mutató alaphelyzetbe állítása.	✓	✓	✓
va_start	Az argumentumlista-mutató átállítása a lista elejére.	✓	✓	✓

F2.3.12. Keresés és rendezés

stdlib h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
bsearch	Bináris keresés tömbben	✓	✓	✓
lfind	Adott érték keresése tömbben lineáris kereséssel.	✓		✓
lsearch	Adott érték keresése lineáris kereséssel, ha nem található, az értéket a tömbhöz hozzácsatolja a függvény	✓		✓
qsort	Tömb elemeinek rendezése a gyorsrendezés (*quicksort*) algoritmusának felhasználásával.	✓	✓	✓

F2.3.13. További szabványos függvények

stdlib h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
getenv	A környezeti változó értékének lekérdezése	✓	✓	✓
_lrotl	**long int** adat bitjeinek eltolása balra	✓		
_lrotr	**long int** adat bitjeinek eltolása jobbra.	✓		
max	Két érték közül a nagyobbat adja (makro).	✓		
min	Két érték közül a kisebbet adja (makro).	✓		
putenv	A környezeti változó értékének beállítása.	✓		✓

Függvény	Leírás	Turbo C	ANSI C	UNIX C
rand	Véletlenszám (egész) előállítása.	✓	✓	✓
random	Egész véletlenszám előállítása 0 és az argumentum-1 között (makro).	✓		
randomize	A pszeudovéletlen számsorozat inicializálása.	✓		
_rotl	**unsigned int** adat bitjeinek forgatása balra.	✓		
_rotr	**unsigned int** adat bitjeinek forgatása jobbra.	✓		
srand	A pszeudovéletlen számsorozat inicializálása.	✓	✓	✓
swab	Két tetszőleges típusú adatelem byte-jainak felcserélése.	✓		✓

F2.3.14. Rendszerhívások

A rendszerhívások csoportjában összefoglaltuk azokat a függvényeket, melyek többsége csak az IBM PC számítógépeken futó C-rendszerekben használatosak. Ezek a hívások lehetővé teszik, hogy a C hatékony rendszerprogramozási eszköz legyen az MS-DOS operációs rendszer alatt is.

F2.3.14.1 BIOS interface

bios.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
biosdisk	A lemezegységek kezelése (a 0x13 megszakítás hívása)	✓		
biosequip	A hardverkörnyezet lekérdezése (a 0x11 megszakítás hívása).	✓		
bioskey	A billentyűzet kezelése (a 0x16 megszakítás hívása).	✓		
biosmemory	Információkérés az elérhető memóriáról (a 0x12 megszakítás hívása)	✓		
biosprint	Adatkivitel a nyomtatóra (a 0x17 megszakítás hívása).	✓		
bioscom	Kommunikáció a soros vonalon (a 0x14 megszakítás hívása).	✓		
biostime	A rendszeróra elérése (a 0x1A megszakítás hívása).	✓		

F2.3.14.2. MS-DOS interface

dos.h, io.h, stdio.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
absread	A diszk abszolút olvasása (a 0x25 megszakítás hívása).	✓		
abswrite	A diszk abszolút írása (a 0x26 megszakítás hívása).	✓		
allocmem	DOS memóriablokk foglalása (MS-DOS 0x48 funkció).	✓		
bdos	MS-DOS rendszerhívás, a DX és az AL regiszterek használatával.	✓		
bdosptr	Rendszerhívás a DS:DX, AL regiszterek használatával.	✓		
country	Az országfüggő információk lekérdezése.	✓		
ctrbrk	A *Ctrl-Break* eseményt kezelő függvény beállítása	✓		
delay	Várakozás adott millisecundum-ig.	✓		
disable	A megszakítások letiltása.	✓		
dosexterr	Az MS-DOS 0x59 megszakítás által visszaadott regiszterérték lekérdezése.	✓		
dostounix	A DOS dátum és idő konvertálása UNIX formátumúvá.	✓		
enable	Megszakítások engedélyezése.	✓		
__emit__	Gépikód elhelyezése a program kódjában.	✓		
FP_OFF	Valamely távoli (**far**) mutató offszet részének lekérdezése (makro).	✓		
FP_SEG	Valamely távoli (**far**) mutató szegmens részének lekérdezése (makro)	✓		
freemem	DOS memóriablokk felszabadítása (MS-DOS 0x49 funkció).	✓		
geninterrupt	Megszakításhívás.	✓		
getcbrk	A *Ctrl-Break* kezelési módjának lekérdezése.	✓		
getcwd	Az aktuális alkönyvtár lekérdezése.	✓		
getdate	A rendszerdátum lekérdezése (MS-DOS 0x2A funkció).	✓		
getdfree	Információkérés az adott lemezegységről (MS-DOS 0x36 funkció)	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
getdta	A DTA (*disk-transfer* cím) lekérdezése.	✓		
getfat	Adott meghajtó FAT (file foglalási tábla) információinak lekérdezése.	✓		
getfatd	Az aktuális meghajtó FAT információinak lekérdezése.	✓		
getftime	Lekérdezi egy file utolsó írásának idejét és dátumát (MS-DOS 0x57 funkció).	✓		
getpsp	A PSP szegmensének lekérdezése.	✓		
gettime	A rendszeridő lekérdezése (MS-DOS 0x2C funkció).	✓		
getvect	A megszakításvektor aktuális értékének lekérdezése (MS-DOS 0x35 funkció)	✓		
getverify	A DOS *verify* állapotjelző lekérdezése.	✓		
harderr	Létrehoz egy hardverhiba-kezelőt.	✓		
hardresume	Hardverhiba esetén visszatér az MS-DOS-hoz.	✓		
hardretn	Hardverhiba lekezelése után visszatér a felhasználói programhoz	✓		
int86	Megszakítások hívása	✓		
int86x	Megszakítások hívása a szegmensregiszterek megadásával	✓		
intdos	MS-DOS funkciók aktivizálása	✓		
intdosx	Ugyanaz, mint az *intdos*, kiegészítve a szegmensregiszterek megadásával.	✓		
intr	80x86 megszakításhívás.	✓		
keep	A program rezidenssé tétele (TSR - MS-DOS 0x31 funkció).	✓		
MK_FP	Távoli (**far**) pointer előállítása szegmens- és offszet címekből (makro)	✓		
nosound	A hangszóró kikapcsolása.	✓		
parsfnm	File-név tesztelése és a file-vezérlő blokk (FCB) felépítése.	✓		
peek	Szó olvasása a memóriából.	✓		
peekb	Byte olvasása a memóriából.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
poke	Szó írása a memóriába.	✓		
pokeb	Byte írása a memóriába.	✓		
randbrd	File-ba írás FCB használattal.	✓		
randbwr	File olvasása FCB-vel.	✓		
setblock	Valamely előzőleg lefoglalt memóriablokk méretének megváltoztatása (MS-DOS 0x4A funkció).	✓		
setcbrk	A *Ctrl-Break* kezelési módjának beállítása..	✓		
setdate	A rendszerdátum beállítása (MS-DOS 0x2B funkció)	✓		
setdta	A DTA beállítása.	✓		
settime	A rendszeridő beállítása (MS-DOS 0x2D funkció)	✓		
setvect	Adott megszakításvektor beállítása (MS-DOS 0x25 funkció)	✓		
setverify	A DOS *verify* állapotjelző beállítása.	✓		
segread	A szegmensregiszterek aktuális értékének lekérdezése	✓		
sleep	Várakozás adott másodpercig.	✓		✓
sound	Hang előállítása adott frekvenciával.	✓		
unixtodos	UNIX időformátum konvertálása DOS formátumúvá.	✓		
unlink	File törlése.	✓		✓

F2.3.15. A BGI grafikus könyvtárhoz tartozó függvények

graphics.h

Függvény	Leírás	Turbo C	ANSI C	UNIX C
arc	Körív rajzolása.	✓		
bar	Kifestett téglalap rajzolása.	✓		
bar3d	Diagramoszlop rajzolása.	✓		
circle	Kör rajzolása.	✓		
cleardevice	A grafikus képernyő törlése.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
clearviewport	Az aktuális *viewport* törlése.	✓		
closegraph	A grafikus üzemmód kikapcsolása.	✓		
detectgraph	Grafikus vezérlő és mód felderítése.	✓		
drawpoly	Poligon rajzolása.	✓		
ellipse	Ellipszis rajzolása.	✓		
fillellipse	Kifestett ellipszis rajzolása.	✓		
fillpoly	Kifestett poligon rajzolása.	✓		
floodfill	Tetszőleges zárt alakzat kifestése	✓		
getarccoords	Ív rajzolása után a jellemző pontok lekérdezése.	✓		
getaspectratio	A képpontarány lekérdezése.	✓		
getbkcolor	Háttérszín lekérdezése.	✓		
getcolor	Aktuális rajzoló szín lekérdezése.	✓		
getdefaultpalette	Az alapértelmezés szerinti paletta lekérdezése.	✓		
getdrivername	A grafikus vezérlő nevének lekérdezése.	✓		
getfillpattern	A kitöltési minta lekérdezése.	✓		
getfillsettings	Az aktuális mintabeállítás lekérdezése.	✓		
getgraphmode	A grafikus mód lekérdezése	✓		
getimage	Képrészlet memóriába töltése	✓		
getlinesettings	Az aktuális vonalbeállítás lekérdezése.	✓		
getmaxcolor	A legnagyobb színkód értéke.	✓		
getmaxmode	A lehetséges grafikus módok száma.	✓		
getmaxx	A maximális vízszintes képernyő koordináta.	✓		
getmaxy	A maximális függőleges képernyő koordináta.	✓		
getmodename	A grafikus mód nevének lekérdezése.	✓		
getmoderange	A lehetséges grafikus módok értékhatárainak lekérdezése.	✓		
getpalette	Az aktuális paletta kiolvasása.	✓		
getpalettesize	Az aktuális paletta méretének lekérdezése.	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
gettextsettings	A szövegírás módjának lekérdezése.	✓		
getviewsettings	Az aktuális *viewport* jellemzőinek lekérdezése	✓		
getx	Az aktuális x-koordináta.	✓		
gety	Az aktuális y-koordináta.	✓		
graphdefaults	Az alapértelmezés szerinti beállítások elvégzése.	✓		
grapherrormsg	A grafikus hibaüzenet szövegének kinyerése.	✓		
_graphfreemem	A grafikus memória felszabadítása.	✓		
_graphgetmem	A grafikus memória lefoglalása.	✓		
graphresult	A grafikus művelet kódja.	✓		
imagesize	A képrészlet méretének számítása.	✓		
initgraph	A grafikus rendszer inicializálása.	✓		
installuserdriver	Új grafikus vezérlő telepítése.	✓		
installuserfont	Új karakterkészlet telepítése	✓		
line	Egyenes rajzolása	✓		
linerel	Egyenes rajzolása (relatív).	✓		
lineto	Egyenes rajzolása (adott pontig)	✓		
moverel	A grafikus kurzor mozgatása (relatív).	✓		
moveto	A grafikus kurzor mozgatása.	✓		
outtext	Szöveg kiírása aktuális pozícióba	✓		
outtextxy	Szöveg kiírása megadott pozícióba.	✓		
pieslice	Kifestett körszelet rajzolása.	✓		
putimage	Képrészlet megjelenítése	✓		
putpixel	Adott színű képpont rajzolása.	✓		
rectangle	Téglalap rajzolása.	✓		
restorecrtmode	Visszatérés a szöveges üzemmódba.	✓		
sector	Kifestett ellipszisszelet rajzolása	✓		
setactivepage	Az aktív képernyőoldal beállítása	✓		

Függvény	Leírás	Turbo C	ANSI C	UNIX C
setallpalette	A teljes paletta beállítása.	✓		
setaspectratio	A képpontarány beállítása.	✓		
setbkcolor	A háttérszín beállítása.	✓		
setcolor	A rajzszín beállítása.	✓		
setfillpattern	Minta definiálása kifestéshez.	✓		
setfillstyle	Minta és szín választása kifestéshez.	✓		
setgraphbufsize	A grafikus puffer méretének beállítása.	✓		
setgraphmode	A grafikus mód beállítása	✓		
setlinestyle	A vonaljellemzők beállítása.	✓		
setpalette	Aktuális paletta választása.	✓		
setrgbpalette	Színek definiálása IBM 8514-es vezérlő esetén.	✓		
settextjustify	A szöveg kiigazításának beállítása	✓		
settextstyle	A szöveg stílusának, irányának és méretének beállítása.	✓		
setusercharsize	Karakterméret definiálása	✓		
setviewport	*Viewport* kijelölése	✓		
setvisualpage	A látható grafikus képernyőlap kiválasztása	✓		
textheight	A kiírandó szöveg magasságának lekérdezése.	✓		
textwidth	A kiírandó szöveg szélességének lekérdezése.	✓		

F2.4. A Turbo C 2.0 nyelv kulcsszavai

+	**asm**		**extern**		**return**
	auto	+	**far**		**short**
	break		**float**	*	**signed**
	case		**for**		**sizeof**
+	**cdecl**		**goto**		**static**
	char	+	**huge**		**struct**
*	**const**		**if**		**switch**
	continue		**int**		**typedef**
	default	+	**interrupt**		**union**
	do		**long**		**unsigned**
	double	+	**near**	*	**void**
	else	+	**pascal**	*	**volatile**
*	**enum**		**register**		**while**

A plusszal (+) jelzett kulcsszavak Turbo C bővítések, míg a csillaggal (*) jelzettek **ANSI C** bővítések. (Az **asm** kulcsszó nem használható a Turbo C integrált fejlesztői környezetében, csak a parancssor TCC fordítóval)

Az alábbi pszeudóváltozók a 80x86 processzor regisztereinek egyszerű elérését teszik lehetővé (TC 2.0 bővítés):

_cs	***_ds***	***_es***	***_ss***
_AX	***_BX***	***_CX***	***_DX***
_BP	***_SP***	***_SI***	***_DI***
_AH	***_AL***	***_BH***	***_BL***
_CH	***_CL***	***_DH***	***_DL***
_FLAGS			

Minden kulcsszó foglalt szó, így azokat nem lehet azonosítóként használni a C programon belül!

F3. Függvények grafikus módban

arc	x,y középpontú körívet rajzol kezdő és végszög között.

```
void far arc(int x, int y, int kezdoszog, int vegszog,
             int sugar);
```

x,y	a kör középpontja,
kezdoszog	a kezdőszög (fokban),
vegszog	a végszög (fokban),
sugar	a kör sugara.

Ha a kezdőszögnek 0 és végszögnek 360 fokot adunk, akkor a függvény teljes kört rajzol

bar	Téglalapot rajzol és befesti adott színnel és mintával.

```
void far bar(int xf, int yf, int xa, int ya);
```

xf, yf	a téglalap bal felső sarka
xa, ya	a téglalap jobb alsó sarka.

bar3d	Téglatestet rajzol, befesti adott színnel és mintával.

```
void far bar3d(int xf, int yf, int xa, int ya, int m,
               int teteje);
```

xf, yf	a téglatest bal felső sarka,
xa, ya	a téglatest jobb alsó sarka,
m	a téglatest mélysége,
teteje	ha nagyobb nullánál, akkor a teteje nyitott, újabb téglatest illeszthető rá, ha nulla, akkor a teteje zárt lesz.

circle	x,y középpontú és r sugarú kört rajzol.

```
void far circle(int x, int y, int r);
```

cleardevice	Törli a grafikus képernyőt.

```
void far cleardevice(void);
```

clearviewport	Törli az aktuális grafikus ablakot.

```
void far clearviewport(void);
```

A grafikus ablak törlése után a grafikus kurzor a (0,0) pozícióba kerül.

closegraph	Lezárja a grafikus ablakot.

```
void far closegraph(void);
```

A grafikus üzemmód inicializálása előtti képernyőt jeleníti meg

detectgraph	Grafikus meghajtót és módot ajánl.

```
void far detectgraph(int far *grmeghajto, int far *grmod);
```

Megvizsgálja a grafikus meghajtót és kiválasztja azt a módot, amely a legjobb felbontást nyújtja. Hiba esetén a *grmeghajto* paraméter -2-vel tér vissza.

F3.1. táblázat

Grafikus meghajtó	numerikus érték
DETECT	0
CGA	1
MCGA	2
EGA	3
EGA64	4
EGAMONO	5
IBM8514	6
HERCMONO	7
ATT400	8
VGA	9
PC3270	10

drawpoly	Pontsorozatot egyenessel köt össze.

```
void far drawpoly(int n, int far *pontok);
```

n a poligon pontpárjainak száma,
pontok az x,y pontpárokat tartalmazó tömbre mutat.

ellipse	x,y középpontú ellipszisívet rajzol.

```
void far ellipse(int x, int y, int kezdoszog, int vegszog,
                 int xtengely, int ytengely);
```

fillellipse	Rajzol és befest egy ellipszist.

```
void far fillellipse(int x, int y,
                     int xtengely, int ytengely);
```

fillpoly	Rajzol és befest egy zárt poligont.

```
void far fillpoly(int n, int far *pontok);
```

n a poligon pontpárjainak száma,
pontok az x,y pontpárokat tartalmazó tömbre mutat.

floodfill	Adott színű vonallal határolt zárt területet fest be.

```
void far floodfill(int x, int y, int szin);
```

x,y a zárt terület egy belső pontjának koordinátái,
szin a határoló vonal színe.

getarccoords	Az utoljára rajzolt ív kezdő- és végkoordinátái.

```
void far getarccoords(struct arccoordstype far
                      *ivkoordinata);
```

ivkoordinata ***arccoordtype*** típusú struktúra, amelyben az utoljára rajzolt ív adatait visszakapjuk.

```
struct arccoordstype(
     int x,y;
     int xstart, ystart, xend, yend;
}
```

ahol

x,y a középpont koordinátái,
xstart, *ystart* az ív kezdőpontjának koordinátái,
xend, *yend* az ív végpontjának koordinátái.

getaspectratio	A pixel ablakainak arányából adódó torzítás helyrehozása.

```
void far getaspectratio(int far *xo, int far *yo);
```

ahol

**xo*, **yo* a pixel ablakainak arányos értéke.

Az *y* arányfaktor (**yo*) minden grafikus meghajtónál a VGA-t kivéve 10000-hez van normálva. Az **xo* (x arányfaktor) kisebb, mint az **yo*, mivel a képpont magassága és szélessége nem egyforma. A VGA vezérlő esetén 'négyzet alakú' a pixel, itt a **xo* egyenlő **yo*-val.

getbkcolor	Megadja a háttér aktuális színét.

```
int far getbkcolor(void)
```

A háttérszín 0-15-ig változhat a grafikus kártyától függően.

getcolor	Megadja a rajzolás (előtér) színét.

```
int far getcolor(void);
```

Visszaadja az utolsó sikeres ***setcolor*** hívás színének értékét (0-15).

getdefaultpalette	A paletta (színskála) értékeivel tér vissza.

```
struct palettetype *far getdefaultpalette(void);
```

A ***palettetype*** típusú struktúrában kapjuk meg az ***initgraph***-ban definiált színskála értékeket.

getdrivername	A grafikus vezérlő nevét adja vissza.

```
char *far getdrivername(void);
```

Az ***initgraph*** aktiválása után egy karakter mutatóval adja vissza az aktív grafikus vezérlő nevét

getfillpattern	A felhasználó által előzőleg definiált mintát adja vissza.

```
void far getfillpattern(char far *festominta);
```

A ***setfillpattern*** által definiált mintát a ***getfillpattern*** betölti egy 8 byte-os területre, ahová a *festominta* mutat.

getfillsettings	Információt ad az aktuális festőmintáról és színéről.

```
void far getfillsettings(struct fillsettingstype
                         far *informacio);
```

F3.2. táblázat

Festőminta neve	Értéke	Leírás
EMPTY_FILL	0	nincs minta, a kitöltés háttér színű
SOLID_FILL	1	egyenletes, halvány tónus
LINE_FILL	2	vízszintes vonalas minta
LTSLASH_FILL	3	dőlt '/' vonalas minta
SLASH_FILL	4	dőlt '/' vastag vonalas minta
BKSLASH_FILL	5	dőlt '\' vastag vonalas minta
LTBKSLASH_FILL	6	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 minta
CLOSE_DOT_FILL	11	közepesen pontozott minta
USER_FILL	12	a felhasználó által definiált minta, amit a ***setfillpattern*** függvénnyel adott meg

A ***getfillsettings*** függvény a ***fillsettingstype*** struktúrát feltölti a jelenlegi festőminta és szín információkkal.

```
struct fillsettingstype {
     int pattern;      /* az aktuális festőminta */
     int color;        /*az aktuális festő szín */
   };
```

getgraphmode	Az aktuális grafikus üzemmódot adja vissza.

```
int far getgraphmode(void);
```

A mód értéke 0-5 között változhat, ami függ az aktuális grafikus kártyától

getimage	A megadott képterület tartalmát a memóriába tölti.

```
void far getimage(int xf, int yf, int xa, int ya,
                  void far *bitterkep);
```

xf, *yf* a tartomány bal felső sarka,
xa, *ya* a tartomány jobb alsó sarka,
bitterkep a pufferre mutató pointer.

getlinesettings	Megadja a vonal típusát, mintáját és vastagságát.

```
void far getlinesettings(struct linesettingstype
                         far *vonalinfo);
```

Betölti a ***linesettingstype*** struktúrát feltölti a vonalhúzáshoz használt beállításokkal.

F3.3. táblázat A vonal típusa

Konstans	Érték	Leírás
SOLID_LINE	0	normál vonal
DOTTED_LINE	1	pontvonal
CENTER_LINE	2	középvonal
DASHED_LINE	3	szaggatott vonal
USERBIT_LINE	4	felhasználó által definiált vonal

F3.4. táblázat A vonal vastagsága

Konstans	Érték	Leírás
NORM_WIDTH	1	normal vonal (1 pont széles)
THICK_WIDTH	3	vastag vonal (3 pont széles)

getmaxcolor	A használható színek maximális számát adja vissza.

```
int far getmaxcolor(void);
```

getmaxmode	A használható grafikus mód maximális számát adja.

```
int far getmaxmode(void);
```

getmaxx	Megadja a használható oszlopok (x) maximális számát.

```
int far getmaxx(void);
```


getmaxy	Megadja a használható sorok (y) maximális számát.

```
int far getmaxy(void);
```

getmodename	Megadja a grafikus mód nevét.

```
char *far getmodename(int modszam);
```

modszam a grafikus mód száma.

getmoderange	Megadja a grafikus meghajtó módjainak számát.

```
void far getmoderange(int meghajto, int far *also,
                      int far *felso);
```

meghajto grafikus vezérlő.

A **also* tartalmazza a grafikus mód alsó, **felso* pedig a felső határát. Hiba esetén mindkét érték -1 lesz.

getpalette	Megadja az aktuális palettát.

```
void far getpalette(struct palettetype far *paletta);
```

A ***palettetype*** struktúra:

```
#define MAXCOLORS 15
struct palettetype {
     unsigned char size;
     signed char colors[MAXCOLORS+1];
     };
```

getpalettesize	A színtábla méretét adja vissza.

```
int far getpalettesize(void);
```

A visszatérési érték az aktuális grafikus módban egyidejűleg felhasználható színek száma.

getpixel	A kijelölt képpont színét adja vissza.

```
unsigned far getpixel(int x, int y);
```

x,*y* a képpont koordinátái

gettextsettings	A használt karakterkészletről ad információt.

```
void far gettextsettings(struct textsettingstype
                         far *szoveginfo);
```

A függvény a ***textsettingstype*** típusú struktúrában tájékoztatást ad a karakterkészlet típusáról, irányáról, méretéről és beállítási helyzetéről.

```
struct textsettingstype {
     int font;          /* karakterkészlet típusa */
     int direction;     /* iránya */
     int charsize;      /* mérete */
     int horiz;         /* vízszintes helyzete */
     int vert;          /* függőleges helyzete */
};
```

getviewsettings	Az aktuális ablak (viewport) adataival tér vissza.

```
void far getviewsettings(struct viewporttype far *ablak);
```

A ***viewporttype*** struktúra:

```
struct viewporttype {
     int left, top;     /* az ablak bal felső sarka */
     int right, bottom; /* az ablak jobb alsó sarka */
     int clip;          /* ablakra vonatkozó vágás */
  };
```

Az ablakra vonatkozó koordináták abszolút képernyő koordinátákban értendők. A *clip* értéke határozza meg az ablakban megjelenő rajz vágását. Ha a *clip* értéke pozitív, akkor van vágás, ebben az esetben a rajznak csak az ablakba eső része látható, nulla esetén nincs vágás.

getx	Az aktuális grafikus pozíció x koordinátáját adja vissza.

```
int far getx(void);
```

A grafikus pozíció *x* koordinátája relatív érték az ablakhoz képest, ami azt jelenti, hogy ha a képernyőn egy ablakot jelöltünk ki, akkor az ahhoz viszonyított koordinátát kapjuk meg

gety	Az aktuális grafikus pozíció y koordinátáját adja vissza.

```
int far gety(void);
```

A grafikus pozíció *y* koordinátája relatív érték az ablakhoz képest.

graphdefaults	A grafikus módot alapértelmezés szerint állítja vissza.

```
void far graphdefaults(void);
```

Az alapállapot beállítása a következő:

- az ablak a teljes képernyőt jelenti,
- a grafikus kurzort (0,0) pozícióba helyezi,
- alapállapotba állítja a grafikus rendszer változóit.

grapherrormsg	Megadja az adott hibakódhoz tartozó hibaüzenet szövegét.

```
char *far grapherrormsg(int hibakod);
```

A *hibakod*-ot a ***graphresult*** függvénnyel kérhetjük le, a ***grapherrormsg*** karakter mutatót ad vissza a hibaüzenet szövegére

graphresult	Utoljára végrehajtott grafikus művelet hibakódját adja

```
int far graphresult(void);
```

A grafikus művelet hibakódjait az F3 5. táblázatban foglaltuk össze.

imagesize	Megadja az adott képterület tárolásához szükséges byte-ok számát.

```
unsigned far imagesize(int xf, int yf, int xa, int ya);
```

xf, *yf* a téglalap bal felső sarka,
xa, *ya* a téglalap jobb alsó sarka.

Ha a tároláshoz nagyobb memória szükséges, mint 64 Kbyte, akkor -1 a visszatérési érték.

F3.5. táblázat

Hiba -kód	Grafikus hibakonstans	Hibaüzenet
0	grOk	Nincs hiba
-1	grNoInitGraph	.BGI file-t nem találja, vagy az InitGraph nincs aktiválva.
-2	grNotDetected	Nem érzékelt grafikus berendezést.
-3	grFileNotFound	Az vezérlő output tartalmú BGI file hiányzik.
-4	grInvalidDriver	Érvénytelen a vezérlő output tartalmú .BGI file.
-5	grNoLoadMem	Kevés a memória a vezérlő betöltéséhez.
-6	grNoScanMem	Kevés a memória.
-7	grNoFloodMem	Kevés a memória.
-8	grFontNotFound	A karakterkészlet file-t nem találja.
-9	grNoFontMem	Kevés a memória a karakterkészlet betöltéséhez.
-10	grInvalidMode	Érvénytelen grafikus mód.
-11	grError	Grafikus hiba.
-12	grIOerror	Grafikus I/O hiba.
-13	grInvalidFont	Érvénytelen karakterkészlet file.
-14	grInvalidFontNum	Érvénytelen karakterkészlet szám.
-15	grInvalidDeviceNum	Érvénytelen a grafikus eszköz száma.
-18	grInvalidVersion	Érvénytelen verziószám.

initgraph	Bekapcsolja a grafikát.

```
void far initgraph(int far *grmeghajto,
                   int far *grmod,
                   char far *utvonal);
```

**grmeghajto*	a grafikus kártya típusa,
**grmod*	grafikus mód,
utvonal	a .BGI file-t tartalmazó könyvtár neve.

A *grmeghajto*=DETECT (CONIO.H) megadása esetén a grafikus kártya típusa és a grafikus mód automatikusan kerül kiválasztásra.
Amennyiben a szükséges .BGI file abban a könyvtárban van, amelyben dolgozunk, akkor a *utvonal* paraméter értéke "" (üres sztring) lehet.

F3.6. táblázat

Grafikus meghajtó	grafikus mód	érték	oszlop x sor felbontás	paletta	lapok száma
CGA	CGAC0	0	320x200	C0	1
	CGAC1	1	320x200	C1	1
	CGAC2	2	320x200	C2	1
	CGAC3	3	320x200	C3	1
	CGAHI	4	640x200	2 szín	1
MCGA	MCGAC0	0	320x200	C0	1
	MCGAC1	1	320x200	C1	1
	MCGAC2	2	320x200	C2	1
	MCGAC3	3	320x200	C3	1
	MCGAMED	4	640x200	2 szín	1
	MCGAHI	5	640x480	2 szín	1
EGA	EGALO	0	640x200	16 szín	4
	EGAHI	1	640x350	16 szín	2
EGA64	EGA64LO	0	640x200	16 szín	1
	EGA64HI	1	640x350	4 szín	1
EGA-MONO	EGAMONOHI	3	640x350	2 szín	1 *
	EGAMONOHI	3	640x350	2 szín	2 **
HERC	HERCMONOHI	0	720x348	2 szín	2
ATT400	ATT400C0	0	320x200	C0	1
	ATT400C1	1	320x200	C1	1
	ATT400C2	2	320x200	C2	1
	ATT400C3	3	320x200	C3	1
	ATT400MED	4	640x200	2 szín	1
	ATT400HI	5	640x400	2 szín	1
VGA	VGALO	0	640x200	16 szín	2
	VGAMED	1	640x350	16 szín	2
	VGAHI	2	640x480	16 szín	1
PC3270	PC3270HI	0	720x350	2 szín	1
IBM8514	IBM8514HI	1	1024x768	256 szín	
	IBM8514LO	0	640x480	256 szín	

* 64K EGAMONO kártyán
** 256K EGAMONO kártyán

installuserdriver	Új grafikus vezérlő program telepítése.

```
int far installuserdriver(char far *nev,
                          int huge (*fnev)(void));
```

nev az új vezérlő program file (.BGI) neve,
fnev az *fnev* egy olyan függvényre mutat, amit akkor aktivizál az ***initgraph***, ha a grafikus mód automatikus detektálását kérjük.

installuserfont	Új karakterkészlet betöltése.

```
int far installuser(char far *utvonal);
```

utvonal az új karakterkészlet file elérési útvonala.

Egyszerre 20 karakterkészlet telepíthető. Hibajelzést kapunk, ha a belső tábla betelik (-11).

line	Egyenest rajzol két pont között.

```
void far line(int x1, int y1, int x2, int y2);
```

x1, *y1* az egyenes kezdő koordináta,
x2, *y2* az egyenes vég koordináta.

Az aktuális színnel, vonaltípussal és vonalvastagsággal egyenest rajzol.

linerel	Egyenest rajzol aktuális pozíciótól dx,dy távolságig.

```
void far linerel(int dx, int dy);
```

dx távolság x irányban,
dy távolság y irányban.

Az egyenest a korábban definiált színnel, vonaltípussal és vonalvastagsággal rajzolja meg.

lineto	Egyenest rajzol aktuális pozíciótól egy adott pontig

```
void far lineto(int x, int y);
```

x, *y* az egyenes végpontja

Az egyenest a korábban definiált színnel, vonaltípussal és vonalvastagsággal rajzolja meg.

moverel	Az aktuális pozíciót (kurzort) adott távolsággal áthelyezi.

```
void far moverel(int dx, int dy);
```

dx relatív távolság x irányban,
dy relatív távolság y irányban.

moveto	Az aktuális pozíciót (kurzort) a megadott pontba mozgatja.

```
void far moveto(int x, int y);
```

x, y a grafikus pointer új helyzetének koordinátái.

outtext	Az aktuális helytől szöveget ír.

```
void far outtext(char far *szoveg);
```

szoveg a kiírandó szöveg mutatója.

outtextxy	Szöveget ír ki a megadott (x,y) pontnál.

```
void far outtextxy(int x, int y, char far *szoveg);
```

x, *y* az adott koordinátapont,
szoveg a kiírandó szöveg mutatója.

pieslice	Rajzol és befest egy körcikket.

```
void far pieslice(int x, int y, int kezdoszog, int vegszog,
                  int sugar);
```

x, *y* a középpont koordinátái,
kezdoszog kezdeti szög,
vegszog a végszög,
sugar a kör sugara

Befest egy (*x*,*y*) középpontú, *kezdoszog* és *vegszog* közötti körcikket az aktuális színnel és mintával. A kezdő- és végszöget az óramutató járásával ellentétes irányban kell megadni.

putimage	Korábban tárolt képmező ráhelyezése a képernyőre.

```
void far putimage(int xf, int yf, void far *bitterkep,
                  int muvelet);
```

xf, yf a téglalap tartomány bal felső sarokpontja a képernyőn,
bitterkep pointer, amely kijelöli a képmezőt tartalmazó területet,
muvelet bináris művelet a kihelyezett tartomány pontjai és a képernyő pontjai között, mely lehetséges értéke:

F3.7. táblázat

Műveleti konstans	Értéke	Leírás
COPY_PUT	0	rámásolja
XOR_PUT	1	kizáró vagy
OR_PUT	2	vagy kapcsolat
AND_PUT	3	és kapcsolat
NOT_PUT	4	a képmező inverzét másolja

putpixel	A megadott pozícióba egy képpontot rajzol.

```
void far putpixel(int x, int y, int szin);
```

Az (x,y) pozícióban képpontot rajzol az adott színnel.

rectangle	Téglalapot rajzol.

```
void far rectangle(int xf, int yf, int xa, int ya);
```

xf, yf téglalap bal felső sarka,
xa, ya téglalap jobb alsó sarka.

registerbgidriver	Grafikus meghajtót regisztrál.

```
int registerbgidriver(void (*driver)(void));
```

A felhasználó által betöltött vagy a programhoz szerkesztett grafikus meghajtót regisztrálja.

registerbgifont	Grafikus karakterkészletet regisztrál.

```
int registerbgifont(void (*font)(void));
```

A felhasználó által betöltött vagy a programhoz szerkesztett karakterkészletet regisztrálja.

restorecrtmode	Visszaállítja a szöveges üzemmódot.

```
void far restorecrtmode(void);
```

Visszaállítja az eredeti szöveges módot, melyet az ***initgraph*** érzékelt A ***restorecrtmode*** visszakapcsolja a grafikus módot szöveges módba. Erre nem alkalmas a ***textmode***, azt csak akkor használjuk, ha a szöveges módban vagyunk és szöveges módot akarunk váltani.

sector	Rajzol és befest egy ellipszis cikket.

```
void far sector(int x, int y, int kezdoszog, int vegszog,
                int xtengely, int ytengely);
```

x, y a középpont koordinátái,
kezdoszog az ív kezdőszöge,
vegszog az ív végszöge,
xtengely a vízszintes tengely hossza,
ytengely a függőleges tengely hossza

setactivepage	Kijelöli az aktív grafikus lapot.

```
void far setactivepage(int lap);
```

lap lap száma.

Több lap használatát csak az EGA (256K), a VGA és a Hercules grafikus kártya teszi lehetővé. A ***setvisualpage*** segítségével váltogathatjuk a látható lapokat A ***setactivepage*** hívással kijelölt lapon mennek végbe a grafikus műveletek.

setallpalette	A teljes paletta színeinek beállítása.

```
void far setallpalette(struct palettetype far *paletta);
```

paletta pointer, amely a ***palettetype*** típusú struktúrára mutat

Az EGA, VGA palettá színeket lehet változtatni a függvénnyel. A ***palettetype*** struktúra:

```
#define MAXCOLORS 15
struct palettetype {
     unsigned char size;
     signed char colors[MAXCOLORS+1];
};
```

F3.8. táblázat

CGA		EGA/VGA	
Név	**Érték**	**Név**	**Érték**
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_BROWN	20
LIGHTGRAY	7	EGA_LIGHTGRAY	7
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

setaspectratio	Beállítja a beépített oldalarányt.

```
void far setaspectratio(int xoldal, int yoldal);
```

xoldal, *yoldal* oldalarányok.

Ha a beépített oldalarányokkal a kör torzult, akkor a hibát szoftver úton kiküszöbölhetjük, ha az oldalarányokat megfelelően megváltoztatjuk.

setbkcolor	Beállítja a háttér színét.

```
void far setbkcolor(int szin);
```

szin a háttér színe lesz, amely lehet számérték (0-15), vagy a szín szimbolikus neve (F3.9. táblázat).

setcolor	Beállítja a rajzolás színét.

```
void far setcolor(int szin);
```

szin a paletta egy színe.

A *szin* paraméter értéke 0-tól ***getmaxcolor*** által visszaadott értékéig változhat, amely a rajzolás színét állítja be.

F3.9. táblázat

Érték	Név	Érték	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

setfillpattern	A felhasználó által definiált festőmintát állítja be.

```
void far setfillpattern(char far *minta, int szin);
```

minta pointer mutat egy 8 byte hosszú memóriaterületre, ez tartalmazza a felhasználó által definiált festőmintát,

szin a festés színe.

setfillstyle	Beállítja az aktuális festőmintát és színt.

```
void far setfillstyle(int minta, int szin);
```

A mintaválasztékot lásd az F3.2. táblázatban.

setgraphbufsize	Változtatja a grafikus pufferterület méretét.

```
unsigned far setgraphbufsize(unsigned n);
```

n a puffer mérete.

A beépített puffer mérete 4096 byte, ha kisebb is elég, akkor memóriaterületet lehet megtakarítani. 650 töréspontú poligon befestéséhez elegendő a beépített buffer

mérete. Ha ennél több töréspontú poligont akarunk befesteni, akkor szükséges a buffer méretét növelni, hogy elkerüljük a buffer túlcsordulását.

setgraphmode	Beállítja a grafikus módot és törli a képernyőt.

```
void far setgraphmode(int mod);
```

mod a beépített grafikus kártya érvényes módja.

setlinestyle	Beállítja az aktuális vonaltípust és vastagságot.

```
void far setlinestyle(int vonaltipus, unsigned minta,
                      int v);
```

vonaltipus a vonal típusa (0 - 4),
minta felhasználói minta,
v a vonal vastagsága.

Ha a *vonaltipus* 0 - 3-ig változik, akkor a minta paraméter 0. Ha a *vonaltipus* értéke 4 (USERBIT_LINE), akkor a minta egy 16 bites hexadecimális szám A vonalszélesség kétfajta lehet, normál vagy vastag.

```
struct linesettingstype {
                    int linestyle;
                    unsigned upattern;
                    int thickness;
                      };
```

F3.10. táblázat A *vonaltipus* paraméter lehetséges értékei

Neve	értéke	leírása
SOLID_LINE	0	teljes vonal
DOTTED_LINE	1	pontozott vonal
CENTER_LINE	2	középvonal
DASHED_LINE	3	szaggatott vonal
USERBIT_LINE	4	bitmintával megadott

F3.11. táblázat A *v* paraméter lehetséges értékei

Neve	értéke	leírása
NORM_WIDTH	1	normál vastag (1 képpont) széles
THICK_WIDTH	2	vastag vonal (3 képpont) széles

setpalette	A palettában adott helyen álló színt lecseréli.

```
void far setpalette(int sorszam, int szin);
```

sorszam a szín sorszáma a táblázatban,
szin a beírandó színkód

Ezzel a függvénnyel lehetőség van a paletta átdefiniálására

settextjustify	A szöveg helyzetének beállítása.

```
void far settextjustify(int v, int f);
```

v vízszintes beállítás,
f függőleges beállítás

A *v* és az *f* paraméterek az alábbi értékeket vehetik fel:

F3.12. táblázat

Neve	Értéke	Leírása	Paraméter használhatja
LEFT_TEXT	0	balra	v
CENTER_TEXT	1	középre	v, f
RIGHT_TEXT	2	jobbra	v
BOTTOM_TEXT	0	aljára	f
TOP_TEXT	2	tetejére	f

settextstyle	Beállítja az aktuális karakterkészlet típusát és méretét.

```
void far settextstyle(int betukeszlet, int irany,
                      int meret);
```

betukeszlet a karakterkészlet neve,
irany az írás iránya,
meret a karakterek nagysága.

F3.13. táblázat A *betukeszlet* paraméter lehetséges értékei

Neve	Értéke	Leírása
DEFAULT_FONT	0	8x8 bitmintájú karakter
TRIPLEX_FONT	1	háromvonalas
SMALL_FONT	2	kisbetű
SANS_SERIF_FONT	3	egyszerű
GOTHIC_FONT	4	gót betű

F3.14. táblázat Az *irany* paraméter lehetséges értékei

Neve	Értéke	Leírása
HORIZ_DIR	0	balról jobbra
VERT_DIR	1	alulról felfelé

setusercharsize	A karakter szélességének és magasságának változtatása.

```
void far setusercharsize(int mx, int dx, int my, int dy);
```

mx, *dx* mx/dx szorozva a beépített karakterszélességével,
my, *dy* my/dy szorozva a beépített karaktermagasságával.

setviewport	Az ablak kijelölése a grafikus képernyőn.

```
void far setviewport(int xf, int yf, int xa, int ya,
                     int vagas);
```

xf, *yf* az ablak bal felső sarka,
xa, *ya* az ablak jobb alsó sarka,
vagas pozitív érték esetén a vágást bekapcsolja,
nulla esetén kikapcsolja.

setvisualpage	Láthatóvá teszi a megadott grafikus lapot.

```
void far setvisualpage(int lap);
```

lap a lap sorszáma.

setwritemode	A vonalhúzás módját állítja be.

```
void far setwritemode(int mod);
```

F3.15. táblázat A *mod* lehetséges értéke

Neve	Értéke	Leírása
COPY_PUT	0	felülírja a képernyőt
XOR_PUT	1	XOR művelet a képernyővel

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. (Csak a ***line***, ***linerel***, ***lineto***, ***rectangle*** és ***drawpoly*** függvényekre van hatással).

textheight	Visszatér a szöveg képpontokban mért magasságával.

```
int far textheight(char far *szoveg);
```

szoveg szövegre mutató pointer.

textwidth	Visszatér a szöveg képpontokban mért szélességével.

```
int far textwidth(char far *szoveg);
```

szoveg szövegre mutató pointer.

F4. A Turbo C fejlesztői rendszer kezelése

Az integrált fejlesztőrendszer indítása a TC paranccsal történik és betöltése után a főmenübe kerülünk. Innen különféle menüpontok kiválasztásával, illetve forró gombbal minden feladat hívható: szövegszerkesztés, fordítás, szimbolikus nyomkövetés, stb.

Az integrált fejlesztőrendszer főmenüjébe az ***F10*** funkciógombbal jutunk. Bármely ponton lehetőségünk van a rendszer segítségét kérni az ***F1*** leütésével. Nagyon hasznos segítőbillentyű a ***Ctrl-F1.*** Ha szövegszerkesztés közben lenyomjuk, akkor a kurzort éppen tartalmazó szóhoz (nyelvi alapelem, függvénynév, stb.) kapcsolódó ismertetés jelenik meg.

Általános érvényű kiléptető az ***ESC*** billentyű. Bármely parancs kiadása közben érvénytelenít, ha pedig valamely almenüben vagyunk, akkor az eggyel magasabb menüszintre vezérel.

F4.1. Szövegszerkesztés

A program beírásához lépjünk be a szövegszerkesztőbe az **Edit** menüpont kiválasztásával. A szövegszerkesztő ú.n. teljes képernyő szerkesztő (*full screen editor*), amelynek kezelését könnyű elsajátítani. A szerkesztés az *Edit* ablakban történik, efölött az említett főmenü, alatta pedig a fordítási - szerkesztési üzenetek ablaka (*message window*) található. A két ablak között az ***F6*** billentyűvel válthatunk. A szerkesztési ablak első sora a státuszsor, ebben az aktuális editálási jellemzőket láthatjuk. Az aktuális file-név NONAME.C (azaz nevenincs), de lemezre tárolva a rendszer kér helyette egy egyedi program nevet.

F4.2. Fordítás, szerkesztés és futtatás

Ha a programunkat beírtuk az *Edit* ablakba, tároljuk el a **File/Save** menüpont kiválasztásával. Ezután le kell fordítanunk (***Alt-F9***) és szerkesztenünk kell. Ez a két lépés igen gyakori, erre az ***F9*** forrógomb (*hot*

key) szolgál. Ha a program beírásakor hibát követtünk el, a hiba oka és helye az üzenet (*Message*) ablakában olvasható. A hibaüzeneten ***Enter***-t ütve felkerülünk a szerkesztési ablakba és a kurzor éppen a hibára (vagy közvetlen környezetére) mutat! A hiba kijavítása után újra az ***F9*** billentyű lenyomásával kérhetjük a futtatható file előállítását.

A hibátlan fordítás esetén futtathatjuk a programot a **Run** (futtatás) menü **Run** parancsával, vagy az ennek megfelelő ***Ctrl-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 Turbo 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. A futási eredmény kiértékelésére az ***Alt-F5*** forró gombbal átkapcsolhatunk a felhasználói képernyőre, bármely billentyű leütésével visszatérhetünk a szerkesztői ablakba.

F4.3. Project fogalma és használata

Nagyobb programok több modulból, azaz több forrás file-ból állnak, ezenkívül általában felhasználásra kerülnek korábban - vagy mások által - készített függvények lefordított formában, tárgykódú file-okként vagy könyvtárakba (.LIB állományba) gyűjtve. A *project file* nagyszámú file együttes figyelését teszi lehetővé, ugyanakkor az a célunk, hogy egy - egy módosítás után csak a szükséges fordításokat kelljen végrehajtani. Egy .H állományra több .C forrás file hivatkozhat. Ha egy .H állomány módosul, akkor ez több .C forrás file újbóli fordítását is szükségessé teheti.

A Turbo C integrált fejlesztői rendszere a projectek kezelését lehetővé teszi. Ezt az információt .PRJ projectleíró file-ba kell tenni, majd a **Project** menüpont segítségével (**Project name**) átadni a rendszernek. A projectleíró file közönséges szöveg file, amit a fejlesztőrendszer szövegszerkesztőjével is megírhatunk és .PRJ file-ba menthetünk ki.

Ha a célprogram csupán PROG1.C és PROG2.C forrás állományokból áll, akkor a megfelelő PROG.PRJ file a következő két sort tartalmazza:

```
prg1.c
prg2.c
```


egyszerűen felsorolva a forrás file-okat. A fejlesztői rendszer a projectleíró file neve után létrehozza a PROG.EXE futtatható file-t. Ehhez két tárgykódú állományt kell a szerkesztőnek átadni: a PROG1.OBJ és a PROG2.OBJ file-okat. A PROG1.OBJ a PROG1.C, a PROG2.OBJ pedig a PROG2.C forrás file lefordításával jön létre. A **Project make** funkció (***F9***) a végeredménytől visszafelé haladva ellenőrzi minden érintett file (PROG1.C, PROG1.OBJ, PROG2.C, PROG2.OBJ) utolsó módosításának időpontját, és összehasonlítja ezeket a PROG.EXE keletkezésének idejével. Ha az utóbbi a legfrissebb, akkor semmit nem kell tenni, sem fordítás, sem szerkesztés nem szükséges. Ezzel az ellenőrzéssel biztosítja a rendszer, hogy csak a szükséges műveletek legyenek végrehajtva.

A projectleíró file tartalmazhat .OBJ és .LIB kiterjesztésű könyvtárakat is:

```
prog1.c
prog2.c
prog3.obj
grafika.lib
```

A másik eszköz az explicit függőség megadásához az, ha egy adott file-hoz felsoroljuk mindazokat az állományokat, amelyeknek megváltozása maga után hozza a .C file újrafordítását. Ha például a PROG1.C file beépít egy PG1.H és egy PG2.H *include* file-t, akkor a projectleíró a következőképpen módosul:

```
prog1.c (pg1.h, pg2.h)
prog2.c
```

Ennek hatására, ha akár a PG1.H, akár a PG2.H megváltozik, akkor a PROG1.C mindenképpen újra fordítódik és a program szerkesztése is végrehajtódik.

Néha előfordul, hogy a file-ok dátumától függetlenül szükséges az összes forrás file-t újrafordítani, azért mert memóriamodellt változtattunk, vagy egyéb fordítási, szerkesztési opciót megváltoztattunk. Ebben az esetben a **Compile** menüpont **Build all** opcióját kiválasztva, a projectleíró alapján, a dátumoktól függetlenül az összes forrás file-t újrafordítja a rendszer, majd összeszerkeszti a futtatható programot.

F5. A lemezmelléklet használata

A lemezen található programokat DOS rendszer alatt a következő módon fejthetjük ki:

Átmegyünk az A meghajtóra:

```
A:
```

A lemezről elindítjuk az INSTALL.EXE programot:

```
A:\>install
```

A telepítő program ezután megjeleníti a lemezen található tömörített file-ok listáját, amelyen kurzor billentyűkkel mozoghatunk, a kifejteni kívánt alkönyvtárakat pedig a ***Szóköz*** (SPACE) billentyű leütésével választhatjuk ki. A kurzor pozíciójánál lévő file tartalmáról rövid összefoglaló jelenik meg a jobb oldali ablakban.

A telepítés ezután az ***ENTER*** leütésével folytatódik a következő kérdéssel:

Hova kéri installálni a példákat?

A válaszként megadott könyvtárba történik a kifejtés, minden tömörített állomány külön alkönyvtárba másolódik.

A lemezeken levő programok három csoportba oszthatók:

- a PELDAKxy nevű állományok az X.Y. alfejezet példáit tartalmazzák,
- a MEGOLDxy nevű file-ok az X.Y. alfejezetben kiírt feladatok megoldását tartalmazzák,
- a MIX állomány pedig "*Ami a könyvből kimaradt* " címet viseli.

F6. Összefoglaló táblázatok

F6.1. A funkcióbillentyűk kódjai

3 : Ctrl-@	66 : F8	106 : Alt-F3
15 : Shift-Tab	67 : F9	107 : Alt-F4
16 : Alt-Q	68 : F10	108 : Alt-F5
17 : Alt-W	71 : Home	109 : Alt-F6
18 : Alt-E	72 : ↑	110 : Alt-F7
19 : Alt-R	73 : PgUp	111 : Alt-F8
20 : Alt-T	75 : ←	112 : Alt-F9
21 : Alt-Y	77 : →	113 : Alt-F10
22 : Alt-U	79 : End	114 : Ctrl-PrintScreen
23 : Alt-I	80 : ↓	115 : Ctrl-←
24 : Alt-O	81 : PgDn	116 : Ctrl-→
25 : Alt-P	82 : Ins	117 : Ctrl-End
30 : Alt-A	83 : Del	118 : Ctrl-PgDn
31 : Alt-S	84 : Shift-F1	119 : Ctrl-Home
32 : Alt-D	85 : Shift-F2	120 : Alt-1
33 : Alt-F	86 : Shift-F3	121 : Alt-2
34 : Alt-G	87 : Shift-F4	122 : Alt-3
35 : Alt-H	88 : Shift-F5	123 : Alt-4
36 : Alt-J	89 : Shift-F6	124 : Alt-5
37 : Alt-K	90 : Shift-F7	125 : Alt-6
38 : Alt-L	91 : Shift-F8	126 : Alt-7
44 : Alt-Z	92 : Shift-F9	127 : Alt-8
45 : Alt-X	93 : Shift-F10	128 : Alt-9
46 : Alt-C	94 : Ctrl-F1	129 : Alt-0
47 : Alt-V	95 : Ctrl-F2	130 : Alt--
48 : Alt-B	96 : Ctrl-F3	131 : Alt-=
49 : Alt-N	97 : Ctrl-F4	132 : Ctrl-PgUp
50 : Alt-M	98 : Ctrl-F5	133 : F11
59 : F1	99 : Ctrl-F6	134 : F12
60 : F2	100 : Ctrl-F7	135 : Shift-F11
61 : F3	101 : Ctrl-F8	136 : Shift-F12
62 : F4	102 : Ctrl-F9	137 : Ctrl-F11
63 : F5	103 : Ctrl-F10	138 : Ctrl-F12
64 : F6	104 : Alt-F1	139 : Alt-F11
65 : F7	105 : Alt-F2	140 : Alt-F12

F6 1 Táblázat
A funkcióbillentyűk visszatérési kódjai

Az IBM PC számítógépen bizonyos billentyűk, illetve billentyűkombinációk két karakterrel térnek vissza. Ezeknél a billentyűknél az első karakter mindig 0.

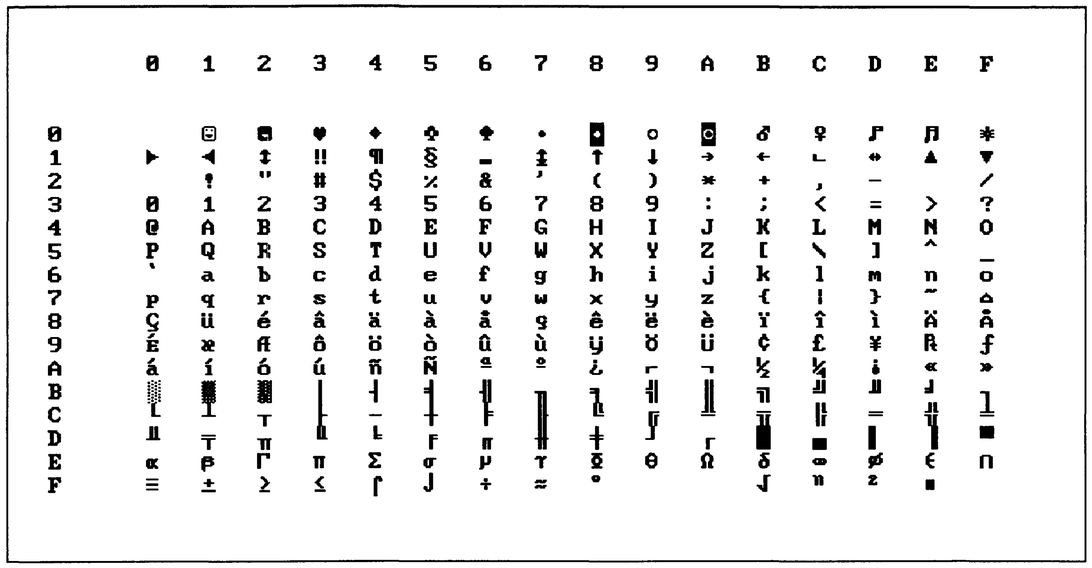
Ez azt jelenti hogy ha a ***getch*** függvény visszatérési értéke 0, akkor funkcióbillentyű lett leütve, és annak kódját egy újabb ***getch*** hívással kapjuk meg.

Az F6.1. táblázat tartalmazza a fent említett billentyűk illetve billentyű-kombinációk második karakterét. Az alábbi program segítségével tetszőleges billentyű-kombinációhoz tartozó kód lekérdezhető:

```
#include <conio.h>

main()
{
   int ch;

   do {
     do; while(!kbhit());
     if (!(ch = getch()))
        ch = getch();
     cprintf("ch:%c - %d\n\r", ch, ch);
   } while (ch != 32);
}
```


F6.2. Az IBM karakterkódok táblázata

A táblázatból hexadecimális formában lehet leolvasni a karakterkódokat. A bal oldali oszlopban található a kód első jegye, míg a felső sorból a második jegy olvasható le. Például az 'A' betű kódja 0x41.



Irodalomjegyzék

B W Kernighan - D M Ritchie
A C programozási nyelv
Műszaki Könyvkiadó, Budapest, 1988

Brian W Kernighan - Dennis M Ritchie
The C programming language (second edition)
Prentice Hall, Inc., New Jersey, 1988

Clovis L Tondo - Scott E Gimpel
C programozási gyakorlatok
Műszaki Könyvkiadó, Budapest, 1988

Pethő Ádám
abC (C programozási nyelvkönyv)
Számítástechnika-alkalmazási Vállalat, Budapest, 1990

Thomas Plum
Tanuljuk meg a C nyelvet
Novotrade Rt., Budapest, 1987

Benkő Tiborné - Poppe András - Benkő László
Bevezetés a Borland C++ programozásba
ComputerBooks, Budapest, 1991

Mark Williams Company
ANSI C. A Lexical Guide
Prentice Hall, New Jersey, 1988

Clauß - Fischer
Programmieren mit C
VEB Verlag Technik, Berlin, 1990

Narain Gehani
C: An Advanced Introduction
Computer Science Press, New Jersey, 1985

Herbert Schild
Using Turbo C
Osborne McGraw-Hill, 1988

M Waite - S Prata - D Martin
C Primer Plus
Howard W Sams and Co, Inc, 1984

Angie Hansen
Learn C now
Microsoft Press, 1988

Kris Jamsa
Microsoft C: secrets, shortcurs, and solutions
Microsoft Press, 1991

Samuel P Harbison - Guy L Steele Jr
C: A Referense Manual
Prentice Hall, New Jersey, 1991

Kenneth A Barclay
ANSI C: Problem-Solving and Programming
Prentice Hall, Inc, 1990

Kocsis Tamás - Poppe András
C programozási feladatgyűjtemény és példatár
BME MTI, Budapest, 1992

Benkő László - Meskó László - Tóth Bertalan - Schuler László
Programozás C és C++ nyelven: példatár
BME MTI, Budapest, 1992

Benkő Tiborné - Urbán Zoltán
IBM PC programozása Turbo C nyelven
BME MTI, Budapest, 1989

Turbo C: User's Guide. Version 2.0
Borland International, 1988

Turbo C: Reference Guide. Version 2.0
Borland International, 1988

Tárgymutató

0 oktális konstans 26
0X hexadecimális konstans 26
+ összeadás operátor 65
& cím operátor 80
= értékadás operátor 78
+= értékadás operátor 78
\\ backslash karakter 28
& bitenkénti ÉS 72
^ bitenkénti kizáró VAGY 72
| bitenkénti VAGY 72
, vessző operátor 82
?: feltételes kifejezés deklaráció 83
-- csökkentő operátor 69
/ osztás operátor 65
== egyenlőség 67
>= nagyobb vagy egyenlő 67
> nagyobb 67
++ növelő operátor 69
* mutató operátor 51,80
!= nem egyenlő 67
<< léptetés balra 76
<= kisebb vagy egyenlő 67
< kisebb 67
&& logikai ÉS 68
! logikai negálás 68
|| logikai VAGY 68
% modulus operátor 65
* szorzás operátora 65
~ egyes komplemens 72
# előfordító operátor 60
előfordító operátor 60
' karakter jelző 27
" karaktersorozat jelző 29
>> jobbra léptetés 76
, struktúra tag operátor 151
-> struktúra pointer 151
- kivonás operátora 65
- mínusz operátor 58,59
+ plusz operátor 59
_ aláhúzás karakter 24
\0 nulla karakter 29

A

abs 304
acos 302
asin 302
atan 302
atan2 302
auto 228,236,238
ANSI 4
  C nyelv kulcsszavai 25
  C szabvány 23
atof 294
atoi 295
atol 295
asctime 317
asszociativitás 61,63
azonosító 24

B

break 6,103,113,114
bsearch 316

C

cabs 304
calloc 313
case 17,101
ceil 305
C előfeldolgozó 248
char 42
clearerr 290
clock 317
continue 6,14,113,115
const 47,48,206
ctime 317
C típusok 39
  alaptípus 39
  származtatott típus 39
coreleft 314
cos 302
cosh 303

D

#define 48,250
definíció 38,46
deklaráció 38 46
  értelmezése 222
difftime 317
div 305
double 29,45
do-while 6,13,111

E,É

ecvt 296
enum 43,44,49,233
#elif 258
#endif 258
#else 258
élettartam 228
  automatikus (lokális) 229
  dinamikus 230
  statikus (globális) 229
#error 262
értékadás 49
  balérték 49
  jobbérték 49
érvényességi tartomány 228,230
expexit 304
extern 36,228,239

F

fabs 305
far 311
fclose 284
fcvt 296
feltételes fordítás 258
feof 287,290
ferrof 290
fflush 15,283
fmod 306
float 29,44
  long 44
  double 44
floor 305
fopen 282
for 6,18,108
_fpreset 306
fprintf 286
fread 287
free 313
fscanf 286
fseek 283,289
fsetpos 283
ftell 289
ftime 317
függvény 173
  argumentuma 186
    const 206
    függvény 186
    struktúra 195
    vektor 197
    kétdimenziós tömb 200
    sztring 201
  attribútumok 177
  deklarációja 179
  hívása 181
  hívási konvenció 177
    cdecl 177
    far 177
    huge 177
    interrupt 177
    near
    pascal
  definíciója 174
  paraméterei 178
  prototípusa 179
  rekurzív 210
  tárolási osztálya 175
  visszatérési típusa 176
függvényre mutató
  pointerek 187
  pointerek tömbje 190
fwrite 289

G

gcvt 297
getc 286
getch 13
getchar 268
gets 15,269,286
gmtime 318
goto 117

H

huge 311

I

if 5,12,95
  else-if 99
  if-else 97
#if 258
#ifdef 258
#ifndef 258
include 33,240,250,261
int 42
  short 43
  long 43
irásjelek 31
isalfa 292
isalnum 292
isascii 292
isntrl 293
isdigit 293
isgraph 293
islower 293
isprint 293
ispunct 293
isspace 293
isupper 293
isxdigit 293
itoa 296

K

kapcsolódás 228,232
kifejezések 58
  kiértékelés menete 91
konstansok 26

L

labs 305
láthatóság 228,230
#line 262
listaszerkezet 166
localtime 318
log 304
log10 304
long 27
  double 29
ltoa 296

M

main 9,33,34
  paraméterei 207
  visszatérési értéke 207
malloc 53,313
megjegyzés 30
memchr 298
memcmp 298
memcpy 298
menicmp 298
memmove 298
memset 298
modf 305
moduláris programozás 35
mutató 50
  dinamikus memóriával 53
  többszörös indirektségű 55

N

near 311
névterületek 233

O

operátor 31,58
  aritmetikai 65
  biteltoló 76
  bitenkénti logikai 72
  elsődleges 65
  értékadó 78
  feltételes 83
  léptető 69
  logikai 67
  összehasonlító 67
  pointer 80
  vessző 82

Q

qsort 316

P

pow 306
#pragma 262
precedencia 61,62
printf 9,33,81,216,270
project file 37
putc 286
putchar 268
puts 268,286

R

rand 306
random 306
randomize 306
realloc 313
register 228,246
rekurzív függvények 210
  önrekurzió 213
  kölcsönös rekurzió 215
return 118
rövidzár kiértékelés 63
rewind 283

S

scanf 11,274
scope 36
setbuf 285
setvbuf 285
sin 302
sinh 304
sizeof 82
sprintf 278
sqrt 306
srand 306
sscanf 15,16,18,278
static 228,243
stdin 15
stime 318
strcat 299
strchr 299
strcmp 299
strcmpi 299
strcpy 202,299
stdup 299
strlen 20,300
strlwr 300
strncat 300
strncmp 300
strpbkr 300
strrchr 300
strrev 300
strset 301
strstr 301
strspn 20
strtod 295
strtol 295
strtok 301
strtoul 295
struct 147,233
  átadása függvénynek 195
  bitmezők használata 163
  egymásba ágyazott 154
  önhivatkozó 166
  struktúratömb 156
tzset 318
switch 5,16,17,101
  default 102

T

tan 302
tanh 304
tárolási osztályok 228
  használata 234
típuskonverzió 85
  implicit 85
    aritmetikai 87
    egész 86
    előjeles 87
    előjel nélküli 87
    lebegőpontos 86
    mutató 88
  explicit 88
típusminősítő 41
típusmódosítás 40
toascii 294
tolower 294
toupper 294
tömb 120
  dinamikus helyfoglalás 136
    egydimenzió esetén 137
    kétdimenzió esetén 139
  egydimenziós 120
    inicializálása 123
    típusdefiniálása 124
  mutató típusú 125,133
    dinamikus helyfogl 142
  többdimenziós 131
  sztrng 127,133
typedef 47,56,147,186,225
  struct esetén 149
  kezdőértékadás 154

U

#undef 257
union 159,233,236
unsigned 27
  long 27
utasítások 93
  blokk 94
ultoa 296

V

va_arg 219
va_end 219
va_list 219
va_start 219
void 54
volatile 41

W

while 6,105

CONTROLL OKTATÓKÖZPONT Kft.

A SZÁMÍTÓGÉP MA MINDENNAPI TÁRGY

Tanulja meg használni!

Szakképesítő tanfolyamok:

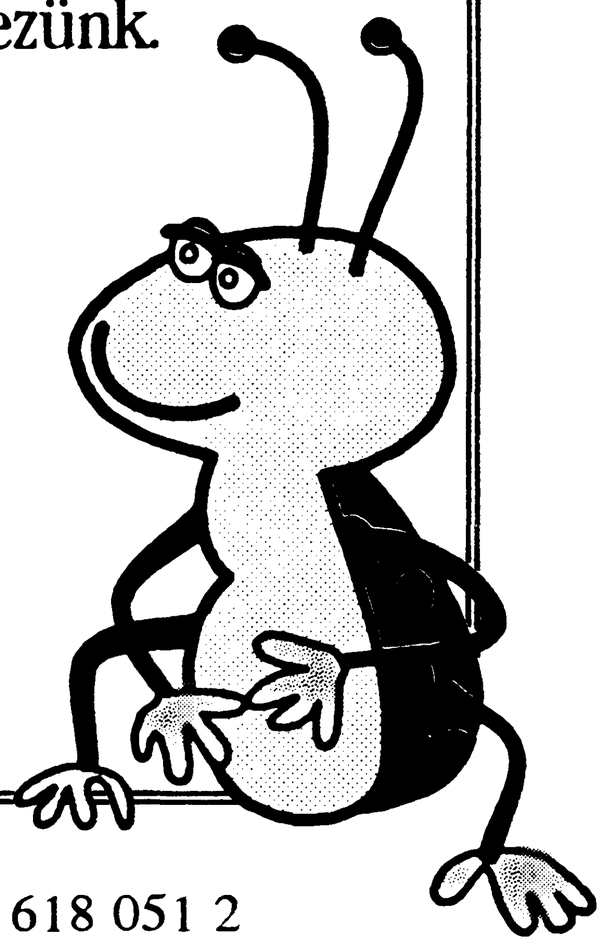
- Számítógép kezelő *(alapfok)*
- Számítástechnikai szoftver üzemeltető *(középfok)*
- Windows az irodában - office *(alapfok)*

Néhány intenzív tanfolyami téma:

- gépkezelés
- szövegszerkesztés
- táblázatkezelés
- hálózatok
- adatbáziskezelés
- grafikus programok
- tanfolyamok vezetőknek, elfoglaltaknak

A tanfolyamokat egyedi tematika szerint ÖNÖKNÉL is megtartjuk.
MUNKANÉLKÜLIEK részére külön tanfolyamokat szervezünk.

KÉRJE RÉSZLETES TÁJÉKOZTATÓINKAT



A *CONTROLL OKTATÓKÖZPONT* a
Microsoft hivatalos oktatóközpontja
1023 Budapest, Csalogány utca 23.
Tel.: 202-6898, 201-6333/511 Fax: 201-7292

ISBN 963 618 051 2

Ára: 1199,— Ft áfával