sapientia
tankönyvek



Kátai Zoltán

Kódolj C-vel!
A programozás alapjai lépésről lépésre

Scientia Kiadó

Kátai Zoltán

Kódolj C-vel! A programozás alapjai lépésről lépésre

Kátai Zoltán

Kódolj C-vel!
A programozás alapjai lépésről lépésre

Scientia Kiadó
Kolozsvár ▪ 2026

sapientia
tankönyvek

informatika

KÉSZÜLT
A MAGYAR KORMÁNY
TÁMOGATÁSÁVAL

TÁRSADALMI KAPCSOLATOKÉRT
ÉS KULTÚRÁÉRT FELELŐS MINISZTÉRIUM

BETHLEN GÁBOR
Alap

DOI: 10.47745/SAPVOL.2026.35

Felelős kiadó:
Sorbán Angella

Lektor:
Kovács Lehel István (Marosvásárhely)

Borítóterv:
Tipotéka Kft.

Kiadói koordinátor:
Szabó Beáta

A szakmai felelősséget teljes mértékben a szerző vállalja.

© Scientia 2026
Minden jog fenntartva, beleértve a sokszorosítás, a nyilvános előadás, a rádió- és televízióadás, valamint a fordítás jogát, az egyes fejezeteket illetően is.

Descrierea CIP a Bibliotecii Naţionale a României
KÁTAI, ZOLTÁN
Kódolj C-vel! : a programozás alapjai lépésről lépésre / Kátai Zoltán. - Cluj-Napoca : Scientia, 2026
Conţine bibliografie
ISBN 978-606-975-122-0

004

Tartalomjegyzék

1. Bevezetés 23
1.1. Az algoritmus mint gondolkodás és mint kultúra 23
1.2. Miért C? – az alap, amelyre minden más épül 23
1.3. A könyv többszálú felépítése 24
1.3.1. A C szál – a programozás szerkezeti alapjai 24
1.3.2. A Hiúz szál – az algoritmikus gondolkodás építése 24
1.3.3. Az AlgoRythmics szál – táncoló algoritmusok 25
1.3.4. A BBC szál – algoritmusok a valóságban 25
1.3.5. Az ECN szál – algoritmika váll váll mellett 25
1.4. AI-cseppek 26
1.5. Kiemelések 26
1.6. A Pareto-elv szellemében épülő könyv 27
1.7. Sapientia – ahol az utak összeérnek 27
2. Algoritmikus gondolkodás és programozás 29
2.1. Hogyan érteti meg magát az ember a géppel 29
2.2. A C nyelv – a programozás közös gyökere 29
2.3. Algoritmus és kód 30
2.4. A B–F–K gondolkodás 30
3. Első lépések: változók, értékadás és I/O-műveletek 32
3.1. Változók – az adatok tárolói a memóriában 32
3.2. Értékadás és aritmetikai operátorok 34
3.3. Standard be- és kimenet 37
3.3.1. A `scanf` függvény 37
3.3.2. A `printf` függvény 38
3.3.3. A `freopen` függvény 39
3.4. Első C programjaink 40
3.5. AlgoRythmics szál: változócsere 43
3.6. BBC szál: csoki vagy csili 45
3.7. Hiúz szál: barlangátkelés 48
3.8. ECN szál: Piroska és a farkas 49
4. Utasításkészlet: kevés darab, végtelen kombináció 52
4.1. A vezérlés három alapformája 52
4.2. A C nyelv alapvető utasításai 52
4.2.1. Kifejezés-utasítás 53
4.2.2. Döntési utasítások – párhuzamos forgatókönyvek 53
4.2.3. Ismétlési utasítások – ugyanaz újra és újra 54
4.2.4. Ugró utasítások – beavatkozás a végrehajtásba 55

5. Elágazás a programban – döntési utasítás 57
5.1. Az `if` utasítás – kétutas elágazás 57
5.2. Egyutas elágazás 59
5.3. Összehasonlítási operátorok 60
5.4. Logikai operátorok 60
5.5. Egymásba ágyazott `if` – több kizáró eset kezelése 62
5.6. BBC szál: csoki vagy csili 2. 64
5.7. Hiúz szál: minimumkeresés négy elemre 65
5.8. AlgoRythmics szál: három szám rendezése – buborék, kiválasztó, beszúró 66
5.9. ECN szál: királyok a sakktáblán. 69
6. Számsorozatok feldolgozása: ismert lépésszámú ismétlés 72
6.1. A `for` ciklus – ismert lépésszámú ismétlés 72
6.1.1. Feldolgozás beolvasás közben 72
6.1.2. Szomszédos elemek kezelése 76
6.2. Egydimenziós tömbök 78
6.2.1. Feldolgozás eltárolt adatokon 79
6.2.2. Számsorozatok generálása és eltárolása 81
6.3. BBC szál: csoki vagy csili 3. 83
6.4. AlgoRythmics szál: lineáris keresés. 85
6.5. Hiúz szál: minimum és maximum keresés tetszőleges elemszámú sorozatban 88
6.6. ECN szál: kétszemélyes játék 90
7. Amíg igaz: feltételek által vezérelt ismétlés 95
7.1. A ciklus-utasítások általános alakja. 95
7.1.1. A `while` ciklus 96
7.1.2. A `do-while` ciklus. 98
7.1.3. A `for` ciklus 99
7.2. Ugró utasítások ciklusokon belül 100
7.2.1. A `break` utasítás. 100
7.2.2. A `continue` utasítás 101
7.3. BBC szál: legnagyobb közös osztó kiszámítása. 103
7.4. AlgoRythmics szál: bináris keresés 105
7.4.1. Bináris versus lineáris keresés 107
7.5. Hiúz szál: prím-teszt 108
7.6. ECN szál: digitális óra 109
8. Pointerek és egydimenziós dinamikus tömbök. 113
8.1. Mutatók 113
8.1.1. Változók címe 113
8.1.2. Pointer deklarációja 114
8.1.3. A `&` és a `*` operátor 115
8.1.4. Több pointer ugyanarra a változóra 116

8.1.5. A NULL pointer 116
8.2. A pointerek és az egydimenziós tömbök kapcsolata 117
8.2.1. A tömb neve és az első elem címe 117
8.2.2. Indexelés pointeren keresztül 118
8.2.3. Pointeraritmetika tömbökben 119
8.2.4. Miben különbözik a tömbnév és a pointer? 119
8.2.5. Tömb bejárása pointerrel 120
8.3. Dinamikus memóriafoglalás egydimenziós tömböknek 121
8.3.1. A malloc és calloc függvények 122
8.3.2. Sikertelen memóriafoglalás 123
8.3.3. Dinamikus tömb beolvasása és kiírása 123
8.3.4. A free függvény 125
8.4. AlgoRythmics szál: két rendezett sorozat összefésülése 126
8.5. Hiúz szál: szétválogatás helyben 129
8.6. BBC szál: az utazóügynök és a mohó útvonal 130
8.7. ECN szál: nagy mátrix feldolgozása két sorral 134
9. Mesterfejezet – összekapcsolás és elmélyítés 138
9.1. Tudjunk meg többet a változókról 138
9.1.1. Változók típusa 139
9.2. Konstansok 142
9.2.1. Egész konstansok 142
9.2.2. Valós konstansok 143
9.2.3. Karakterkonstansok 143
9.2.4. Karakterlánc-konstansok 144
9.2.5. Szimbolikus konstansok 144
9.2.6. Lebegőpontos csapdák 145
9.3. Formátumkarakterek a scanf és printf használatában 146
9.3.1. Formázott kiírás – mezőszélesség és pontosság 148
9.4. Tömbök 150
9.4.1. Egydimenziós tömbök rövid áttekintése 150
9.4.2. Kétdimenziós tömbök 151
9.4.2.1. Kétdimenziós tömbök feldolgozása 152
9.4.2.2. Szimmetriák négyzetes mátrixokban 153
9.4.3. Többdimenziós tömbök (*n*-dimenziós általánosítás) 155
9.5. A pointerek haladóbb használata 156
9.5.1. Pointerek explicit típuskonverziója 156
9.5.2. A void * pointer 158
9.5.3. A const és a pointerek 159
9.5.4. Kétdimenziós tömbök és pointerek 161
9.5.5. Dinamikusan létrehozott kétdimenziós tömbök 162
10. Mesterfejezet: kifejezések és vezérlés – a C valódi arca 166
10.1. Kifejezések 166

10.1.1. Operandusok és operátorok 166
10.1.2. Operátorprecedencia és zárójelezés 167
10.1.3. A kifejezések értéke és típusa 167
10.1.3.1. Implicit típuskonverzió 168
10.1.3.2. Explicit típuskonverzió 169
10.1.4. Aritmetikai operátorok 169
10.1.4.1. Inkrementáló és dekrementáló operátorok 169
10.1.5. Relációs és logikai kifejezések 170
10.1.6. A feltételes operátor (?:) 171
10.1.7. A vessző operátor (,) 172
10.1.8. Az értékadás mint kifejezés 172
10.2. Bitműveletek és bitmaszkok 174
10.2.1. Bitenkénti logikai operátorok 174
10.2.2. Bitenkénti negálás 175
10.2.3. Biteltoló operátorok 176
10.2.4. Bitmaszkok 177
10.2.5. Több állapot tárolása egyetlen értékben 179
10.2.6. Összetett bitműveleti értékadások 180
10.2.7. Precedencia és zárójelezés 181
10.3. Kifejezés-utasítás 183
10.3.1. A kifejezés-utasítás lényege: a hatás, nem az érték 183
10.3.2. Milyen kifejezések alkalmasak utasításnak? 184
10.3.3. Függvényhívás mint kifejezés-utasítás 184
10.3.4. Kifejezés-utasítás és vezérlés 185
10.3.5. Túl tömör kifejezés-utasítás veszélye 185
10.3.6. Üres és összetett utasítás – szintaktikai kényszer, szemantikai szabadság 186
10.4. Értékvezérelt elágazás: a `switch` utasítás 187
10.4.1. A `switch` szerkezet 187
10.4.2. A „`switch`-`break`” és a fall-through jelenség 189
10.4.3. A `switch` korlátai 190
10.5. Az ismétlés mint vezérlési modell 191
10.5.1. Feltételvezérelt ismétlés 191
10.5.2. Eseményvezérelt ismétlés 191
10.5.3. Menüvezérelt program: minikalkulátor két szemléletben 192
10.5.4. Többszörös egymásba ágyazás 194
11. Függvények: részfeladatok saját kézben 196
11.1. Függvénydefiníció, függvénydeklaráció és függvényhívás 197
11.2. Paraméterátadás 202
11.3. Tömbök átadása függvényeknek 206
11.3.1. Egydimenziós tömbök mint függvényparaméterek 206
11.3.2. Kétdimenziós tömbök átadása függvényeknek 208

11.4. BBC szál: Sudoku-ellenőrző 210
11.5. AlgoRythmics szál: buborékrendezés 212
11.6. Hiúz szál: hódok és viperák 215
11.7. ECN szál: hegylánc a mátrixban 216
12. Rekurzió: amikor a függvény önmagát hívja 218
12.1. Rekurzió: a ciklus alternatívája 218
12.2. A rekurzió oda-vissza útja 220
12.3. Rekurzív függvények, amelyek értéket számítanak ki 223
12.3.1. Amikor már van rekurzív képlet 223
12.3.2. Számjegyenkénti feldolgozás 224
12.3.3. Számsorozat elemenkénti feldolgozása 225
12.4. AlgoRythmics szál: keresés és rendezés rekurzívan 227
12.4.1. Bináris keresés rekurzívan 227
12.4.2. Buborékrendezés rekurzívan 228
12.4.3. Amikor a rekurzió természetes: összefésüléses rendezés 229
12.5. BBC szál: PageRank – amikor a fontosság mások fontosságától függ . . 231
12.6. Hiúz szál: kijut-e a hiúz a labirintusból? 233
12.7. ECN szál: függőségek – mit kell előbb létrehozni? 235
13. Karakterek és karakterláncok 239
13.1. Karakterek: amikor a betű mögött szám áll 239
13.2. Karakterláncok: szöveg mint karaktertömb 241
13.2.1. Karakterláncok beolvasása és kiírása 241
13.2.2. Műveletek karakterláncokkal 243
13.2.3. A `<string.h>` könyvtár 245
13.3. Szöveges állomány feldolgozása karakterenként, szavanként és soronként 247
13.4. AlgoRythmics szál: névsor építése beszúró rendezéssel 250
13.5. BBC szál: stabil házasság probléma – nevek, listák, párosítások 254
13.6. Hiúz szál: Caesar-titkosítás 257
13.7. ECN szál: karakterlánc egyszerűsítése 258
14. Struktúrák: amikor több adat egy objektumhoz tartozik 262
14.1. Az első struktúra 262
14.2. Mezők beolvasása és kiírása 263
14.3. Struktúra különböző típusú mezőkkel 264
14.4. Struktúrák tömbje 265
14.5. Struktúra átadása függvénynek 266
14.6. Struktúrák és `typedef` 269
14.7. Összetartozó adatok egyben 270
14.8. AlgoRythmics szál: kiválasztásos rendezés struktúrákra 272
14.9. BBC szál: ütemezés – amikor a sorrend számít 275
14.10. ECN szál: hatékony rendezés és keresés könyvtári függvényekkel . . 279
14.11. Hiúz szál: házasságok és válások nyilvántartása 282

15. Állománykezelés: amikor az adatok megmaradnak 286
15.1. Állományok megnyitása és lezárása 286
15.2. Szöveges állományok 287
15.3. Szöveges és bináris állományok 289
15.4. Írás bináris állományba: `fwrite` 290
15.5. Olvasás bináris állományból: `fread` 291
15.6. Struktúra kiírása bináris állományba 292
15.7. Struktúratömb mentése és visszaolvasása 292
15.8. Közvetlen hozzáférés bináris állományban 295
15.9. AlgoRythmics szál: gyorsrendezés bináris állományból beolvasott rekordokon 296
15.10. BBC szál: arcfelismerés, képfeldolgozás és bináris képállományok. . 300
15.11. Hiúz szál: egyensúlyi pontok a vadkamera felvételeiben 303
15.12. ECN szál: amikor a beolvasás és kiírás a feladat része 305
16. Mesterfejezet: haladó eszközök, mélyebb rétegek 310
16.1. Függvénypointerek: amikor műveletet adunk át 310
16.2. Változó paraméterszámú függvények 313
16.3. A `main` paraméterei 313
16.4. Indirekt rekurzió 314
16.5. Karakterláncok: sorok, mezők és memóriaterületek 315
16.6. Típusok szervezése: `typedef`, `enum`, `union` 317
16.7. Állománykezelés: karakterek, sorok, pozíciók 319
16.8. Az előfordító szerepe: `include`, `define` és makrók. 320
17. C programozás madártávlatból: a szálak összeérnek 324
17.1. C szál: a program belső rendje 325
17.1.1. Az algoritmustól a programig 325
17.1.2. A program mint szerkezet 326
17.1.3. A típusok összképe 327
17.1.4. Globális, lokális, statikus és dinamikus változók 328
17.2. AlgoRythmics szál: algoritmusok mozgásban 331
17.3. BBC szál: algoritmusok a világ kulisszái mögött 331
17.4. Hiúz szál: éleslátás a kód előtt 332
17.5. ECN szál: próbára tett algoritmusok 333
Irodalomjegyzék 335
Rezumat. Codează în C! – Bazele programării, pas cu pas 337
Abstract. Code in C! – The Fundamentals of Programming, Step by Step . . 338
A szerzőről 339

Cuprins

1. Introducere 23
1.1. Algoritmul ca mod de gândire şi ca element de cultură. 23
1.2. De ce C? – fundamentul pe care se construieşte totul 23
1.3. Structura pe mai multe fire a cărţii 24
1.3.1. Firul C – bazele structurale ale programării. 24
1.3.2. Firul Râsului – dezvoltarea gândirii algoritmice 24
1.3.3. Firul AlgoRythmics – algoritmi dansaţi 25
1.3.4. Firul BBC – algoritmi în lumea reală 25
1.3.5. Firul ECN – algoritmică umăr la umăr 25
1.4. Picături de AI 26
1.5. Evidenţieri 26
1.6. O carte construită în spiritul principiului Pareto 27
1.7. Sapientia – locul unde drumurile se întâlnesc 27
2. Gândire algoritmică şi programare 29
2.1. Cum comunică omul cu maşina 29
2.2. Limbajul C – rădăcina comună a programării. 29
2.3. Algoritm şi cod 30
2.4. Gândirea I–P–O 30
3. Primii paşi: variabile, atribuire şi operaţii de I/O 32
3.1. Variabilele – spaţiile de stocare a datelor în memorie 32
3.2. Atribuirea şi operatorii aritmetici 34
3.3. Intrarea şi ieşirea standard. 37
3.3.1. Funcţia `scanf` 37
3.3.2. Funcţia `printf` 38
3.3.3. Funcţia `freopen` 39
3.4. Primele noastre programe C. 40
3.5. Firul AlgoRythmics: interschimbarea variabilelor 43
3.6. Firul BBC: ciocolată sau chili 45
3.7. Firul Râsului: traversarea peşterii 48
3.8. Firul ECN: Scufiţa Roşie şi lupul. 49
4. Setul de instrucţiuni: puţine elemente, combinaţii infinite. 52
4.1. Cele trei forme fundamentale ale controlului. 52
4.2. Instrucţiunile de bază ale limbajului C. 52
4.2.1. Instrucţiunea-expresie 53
4.2.2. Instrucţiuni de decizie – scenarii paralele. 53
4.2.3. Instrucţiuni repetitive – acelaşi lucru din nou şi din nou 54
4.2.4. Instrucţiuni de salt – intervenţie în execuţie. 55
5. Ramificaţia în program – instrucţiunea de decizie 57
5.1. Instrucţiunea `if` – ramificaţie pe două căi 57

5.2. Ramificaţia pe o singură cale . . . 59
5.3. Operatorii de comparaţie . . . 60
5.4. Operatorii logici . . . 60
5.5. Instrucţiuni `if` imbricate – gestionarea mai multor cazuri mutual exclusive . . . 62
5.6. Firul BBC: ciocolată sau chili – 2 . . . 64
5.7. Firul Râsului: căutarea minimului pentru patru elemente . . . 65
5.8. Firul AlgoRythmics: sortarea a trei numere – prin bule, prin selecţie, prin inserţie . . . 66
5.9. Firul ECN: regi pe tabla de şah . . . 69
6. Prelucrarea şirurilor de numere: repetare cu număr cunoscut de paşi . . . 72
6.1. Ciclul `for` – repetare cu număr cunoscut de paşi . . . 72
6.1.1. Prelucrarea în timpul citirii . . . 72
6.1.2. Prelucrarea elementelor învecinate . . . 76
6.2. Tablouri unidimensionale . . . 78
6.2.1. Prelucrarea datelor stocate . . . 79
6.2.2. Generarea şi stocarea şirurilor de numere . . . 81
6.3. Firul BBC: ciocolată sau chili – 3 . . . 83
6.4. Firul AlgoRythmics: căutarea liniară . . . 85
6.5. Firul Râsului: căutarea minimului şi maximului într-un şir cu un număr arbitrar de elemente . . . 88
6.6. Firul ECN: joc pentru doi jucători . . . 90
7. Cât timp este adevărat: repetare controlată de condiţii . . . 95
7.1. Forma generală a instrucţiunilor repetitive . . . 95
7.1.1. Ciclul `while` . . . 96
7.1.2. Ciclul `do-while` . . . 98
7.1.3. Ciclul `for` . . . 99
7.2. Instrucţiuni de salt în interiorul ciclurilor . . . 100
7.2.1. Instrucţiunea `break` . . . 100
7.2.2. Instrucţiunea `continue` . . . 101
7.3. Firul BBC: calcularea celui mai mare divizor comun . . . 103
7.4. Firul AlgoRythmics: căutarea binară . . . 105
7.4.1. Căutarea binară versus căutarea liniară . . . 107
7.5. Firul Râsului: test de primalitate . . . 108
7.6. Firul ECN: ceas digital . . . 109
8. Pointeri şi tablouri dinamice unidimensionale . . . 113
8.1. Pointeri . . . 113
8.1.1. Adresa variabilelor . . . 113
8.1.2. Declararea unui pointer . . . 114
8.1.3. Operatorii `&` şi `*` . . . 115
8.1.4. Mai mulţi pointeri către aceeaşi variabilă . . . 116
8.1.5. Pointerul `NULL` . . . 116

8.2. Relaţia dintre pointeri şi tablourile unidimensionale. 117
8.2.1. Numele tabloului şi adresa primului element 117
8.2.2. Indexarea prin intermediul pointerilor 118
8.2.3. Aritmetica pointerilor în tablouri. 119
8.2.4. Prin ce diferă numele unui tablou de un pointer?. 119
8.2.5. Parcurgerea unui tablou cu ajutorul unui pointer 120
8.3. Alocarea dinamică a memoriei pentru tablouri unidimensionale 121
8.3.1. Funcţiile `malloc` şi `calloc` 122
8.3.2. Alocarea nereuşită a memoriei. 123
8.3.3. Citirea şi afişarea unui tablou dinamic 123
8.3.4. Funcţia `free` 125
8.4. Firul AlgoRythmics: interclasarea a două şiruri ordonate 126
8.5. Firul Râsului: partiţionare in-place 129
8.6. Firul BBC: comis-voiajorul şi traseul greedy. 130
8.7. Firul ECN: prelucrarea unei matrice mari folosind două linii 134
9. Capitol de aprofundare – integrare şi aprofundare 138
9.1. Să aflăm mai multe despre variabile. 138
9.1.1. Tipul variabilelor 139
9.2. Constante 142
9.2.1. Constante întregi. 142
9.2.2. Constante reale 143
9.2.3. Constante caracter. 143
9.2.4. Constante şir de caractere. 144
9.2.5. Constante simbolice 144
9.2.6. Capcanele numerelor în virgulă mobilă. 145
9.3. Specificatori de format în utilizarea scanf şi printf 146
9.3.1. Afişarea formatată – lăţimea câmpului şi precizia 148
9.4. Tablouri. 150
9.4.1. Scurtă prezentare a tablourilor unidimensionale 150
9.4.2. Tablouri bidimensionale. 151
9.4.2.1. Prelucrarea tablourilor bidimensionale 152
9.4.2.2. Simetrii în matrice pătratice 153
9.4.3. Tablouri multidimensionale (generalizarea la n dimensiuni) . . 155
9.5. Utilizarea avansată a pointerilor 156
9.5.1. Conversia explicită a tipului pointerilor 156
9.5.2. Pointerul `void*` 158
9.5.3. `const` şi pointerii 159
9.5.4. Tablourile bidimensionale şi pointerii. 161
9.5.5. Tablouri bidimensionale create dinamic 162
10. Capitol de aprofundare: expresii şi control – adevărata faţă a limbajului C 166
10.1. Expresii. 166

10.1.1. Operanzi şi operatori 166
10.1.2. Precedenţa operatorilor şi utilizarea parantezelor 167
10.1.3. Valoarea şi tipul expresiilor 167
10.1.3.1. Conversia implicită de tip 168
10.1.3.2. Conversia explicită de tip 169
10.1.4. Operatori aritmetici 169
10.1.4.1. Operatorii de incrementare şi decrementare 169
10.1.5. Expresii relaţionale şi logice 170
10.1.6. Operatorul condiţional (?:) 171
10.1.7. Operatorul virgulă (,) 172
10.1.8. Atribuirea ca expresie 172
10.2. Operaţii pe biţi şi măşti de biţi 174
10.2.1. Operatori logici pe biţi 174
10.2.2. Negarea pe biţi 175
10.2.3. Operatorii de deplasare a biţilor 176
10.2.4. Măşti de biţi 177
10.2.5. Stocarea mai multor stări într-o singură valoare 179
10.2.6. Atribuiri compuse cu operaţii pe biţi 180
10.2.7. Precedenţă şi utilizarea parantezelor 181
10.3. Instrucţiunea-expresie 183
10.3.1. Esenţa instrucţiunii-expresie: efectul, nu valoarea 183
10.3.2. Ce expresii pot fi utilizate ca instrucţiuni? 184
10.3.3. Apelul unei funcţii ca instrucţiune-expresie 184
10.3.4. Instrucţiunea-expresie şi controlul 185
10.3.5. Pericolul unei instrucţiuni-expresie excesiv de compactă 185
10.3.6. Instrucţiunea vidă şi instrucţiunea compusă – constrângere sintactică, libertate semantică 186
10.4. Ramificaţie controlată de valoare: instrucţiunea `switch` 187
10.4.1. Structura `switch` 187
10.4.2. „`switch-break`” şi fenomenul fall-through 189
10.4.3. Limitările instrucţiunii `switch` 190
10.5. Repetarea ca model de control 191
10.5.1. Repetare controlată de condiţie 191
10.5.2. Repetare controlată de evenimente 191
10.5.3. Program controlat prin meniu: mini-calculator în două abordări . . 192
10.5.4. Imbricări multiple 194
11. Funcţii: subproblemele în propriile noastre mâini 196
11.1. Definiţia, declararea şi apelul funcţiilor 197
11.2. Transmiterea parametrilor 202
11.3. Transmiterea tablourilor către funcţii 206
11.3.1. Tablourile unidimensionale ca parametri ai funcţiilor 206
11.3.2. Transmiterea tablourilor bidimensionale către funcţii 208

11.4. Firul BBC: verificarea unui Sudoku 210
11.5. Firul AlgoRythmics: sortarea prin bule 212
11.6. Firul Râsului: castori şi vipere 215
11.7. Firul ECN: lanţ muntos în matrice 216
12. Recursivitate: când funcţia se apelează pe ea însăşi 218
12.1. Recursivitatea: alternativa ciclurilor 218
12.2. Drumul dus-întors al recursivităţii 220
12.3. Funcţii recursive care calculează o valoare 223
12.3.1. Când avem deja o formulă recursivă 223
12.3.2. Prelucrarea cifră cu cifră 224
12.3.3. Prelucrarea element cu element a unui şir de numere 225
12.4. Firul AlgoRythmics: căutare şi sortare recursivă 227
12.4.1. Căutarea binară recursivă 227
12.4.2. Sortarea prin bule recursivă 228
12.4.3. Când recursivitatea este naturală: sortarea prin interclasare . . . 229
12.5. Firul BBC: PageRank – când importanţa depinde de importanţa altora 231
12.6. Firul Râsului: poate râsul să iasă din labirint? 233
12.7. Firul ECN: dependenţe – ce trebuie creat mai întâi 235
13. Caractere şi şiruri de caractere 239
13.1. Caractere: când în spatele literei se află un număr 239
13.2. Şiruri de caractere: textul ca tablou de caractere 241
13.2.1. Citirea şi afişarea şirurilor de caractere 241
13.2.2. Operaţii cu şiruri de caractere 243
13.2.3. Biblioteca `<string.h>` 245
13.3. Prelucrarea fişierelor text caracter cu caracter, cuvânt cu cuvânt şi linie cu linie 247
13.4. Firul AlgoRythmics: construirea unei liste de nume prin sortare prin inserţie 250
13.5. Firul BBC: problema căsătoriilor stabile – nume, liste, asocieri 254
13.6. Firul Râsului: cifrul Caesar 257
13.7. Firul ECN: simplificarea unui şir de caractere 258
14. Structuri: când mai multe date aparţin aceluiaşi obiect 262
14.1. Prima structură 262
14.2. Citirea şi afişarea câmpurilor 263
14.3. Structură cu câmpuri de tipuri diferite 264
14.4. Tablou de structuri 265
14.5. Transmiterea unei structuri către o funcţie 266
14.6. Structuri şi `typedef` 269
14.7. Date conexe păstrate împreună 270
14.8. Firul AlgoRythmics: sortare prin selecţie pentru structuri 272
14.9. Firul BBC: planificare – când ordinea contează 275

14.10. Firul ECN: sortare şi căutare eficientă cu funcţii de bibliotecă 279
14.11. Firul Râsului: evidenţa căsătoriilor şi divorţurilor 282
15. Lucrul cu fişiere: când datele se păstrează 286
15.1. Deschiderea şi închiderea fişierelor 286
15.2. Fişiere text 287
15.3. Fişiere text şi binare 289
15.4. Scrierea într-un fişier binar: `fwrite` 290
15.5. Citirea dintr-un fişier binar: `fread` 291
15.6. Scrierea unei structuri într-un fişier binar 292
15.7. Salvarea şi recitirea unui tablou de structuri 292
15.8. Acces direct într-un fişier binar 295
15.9. Firul AlgoRythmics: sortare rapidă a înregistrărilor citite dintr-un fişier binar 296
15.10. Firul BBC: recunoaştere facială, prelucrarea imaginilor şi fişiere binare de imagine 300
15.11. Firul Râsului: puncte de echilibru în imaginile camerelor de monitorizare a faunei 303
15.12. Firul ECN: când citirea şi afişarea fac parte din problemă 305
16. Capitol de aprofundare: instrumente avansate, niveluri mai profunde .. 310
16.1. Pointeri la funcţii: când transmitem o operaţie 310
16.2. Funcţii cu număr variabil de parametri 313
16.3. Parametrii funcţiei `main` 313
16.4. Recursivitate indirectă 314
16.5. Şiruri de caractere: linii, câmpuri şi zone de memorie 315
16.6. Organizarea tipurilor: `typedef`, `enum`, `union` 317
16.7. Lucrul cu fişiere: caractere, linii, poziţii 319
16.8. Rolul preprocesorului: `include`, `define` şi macrocomenzi 320
17. Programarea în C văzută de sus: firele se întâlnesc 324
17.1. Firul C: ordinea internă a programului 325
17.1.1. De la algoritm la program 325
17.1.2. Programul ca structură 326
17.1.3. Imaginea de ansamblu a tipurilor 327
17.1.4. Variabile globale, locale, statice şi dinamice 328
17.2. Firul AlgoRythmics: algoritmi în mişcare 331
17.3. Firul BBC: algoritmi în culisele lumii 331
17.4. Firul Râsului: vedere ageră înainte de cod 332
17.5. Firul ECN: algoritmi puşi la încercare 333
Bibliografie 335
Rezumat. Codează în C! – Bazele programării, pas cu pas 337
Despre autor 339

Contents

1. Introduction 23
1.1. The Algorithm as a Way of Thinking and as Culture 23
1.2. Why C? – The Foundation on Which Everything Else Is Built 23
1.3. The Multi-Threaded Structure of the Book 24
1.3.1. The C Thread – Structural Foundations of Programming 24
1.3.2. The Lynx Thread – Building Algorithmic Thinking 24
1.3.3. The AlgoRythmics Thread – Dancing Algorithms 25
1.3.4. The BBC Thread – Algorithms in the Real World 25
1.3.5. The ECN Thread – Algorithmics Shoulder to Shoulder 25
1.4. AI Drops 26
1.5. Highlights 26
1.6. A Book Built in the Spirit of the Pareto Principle 27
1.7. Sapientia – Where the Paths Converge 27
2. Algorithmic Thinking and Programming 29
2.1. How Humans Make Themselves Understood by Machines 29
2.2. The C Language – The Common Root of Programming 29
2.3. Algorithm and Code 30
2.4. I–P–O Thinking 30
3. First Steps: Variables, Assignment, and I/O Operations 32
3.1. Variables – Data Containers in Memory 32
3.2. Assignment and Arithmetic Operators 34
3.3. Standard Input and Output 37
3.3.1. The `scanf` Function 37
3.3.2. The `printf` Function 38
3.3.3. The `freopen` Function 39
3.4. Our First C Programs 40
3.5. AlgoRythmics Thread: Swapping Variables 43
3.6. BBC Thread: Chocolate or Chili 45
3.7. Lynx Thread: Cave Crossing 48
3.8. ECN Thread: Little Red Riding Hood and the Wolf 49
4. The Instruction Set: Few Elements, Infinite Combinations 52
4.1. The Three Fundamental Forms of Control 52
4.2. Basic Statements of the C Language 52
4.2.1. Expression Statement 53
4.2.2. Selection Statements – Parallel Scenarios 53
4.2.3. Iteration Statements – The Same Thing Again and Again 54
4.2.4. Jump Statements – Intervening in Execution 55

5. Branching in a Program – Selection Statements 57
5.1. The `if` Statement – Two-Way Branching 57
5.2. One-Way Branching 59
5.3. Comparison Operators 60
5.4. Logical Operators 60
5.5. Nested `if` Statements – Handling Multiple Mutually Exclusive Cases . 62
5.6. BBC Thread: Chocolate or Chili – 2 64
5.7. Lynx Thread: Finding the Minimum of Four Elements 65
5.8. AlgoRythmics Thread: Sorting Three Numbers – Bubble, Selection, and Insertion Sort 66
5.9. ECN Thread: Kings on a Chessboard 69
6. Processing Number Sequences: Iteration with a Known Number of Steps . .72
6.1. The for Loop – Iteration with a Known Number of Steps 72
6.1.1. Processing During Input 72
6.1.2. Processing Adjacent Elements 76
6.2. One-Dimensional Arrays 78
6.2.1. Processing Stored Data 79
6.2.2. Generating and Storing Number Sequences 81
6.3. BBC Thread: Chocolate or Chili – 3 83
6.4. AlgoRythmics Thread: Linear Search 85
6.5. Lynx Thread: Finding the Minimum and Maximum in a Sequence of Arbitrary Length 88
6.6. ECN Thread: Two-Player Game 90
7. While True: Condition-Controlled Iteration 95
7.1. General Form of Loop Statements 95
7.1.1. The `while` Loop 96
7.1.2. The `do-while` Loop 98
7.1.3. The `for` Loop 99
7.2. Jump Statements Within Loops 100
7.2.1. The `break` Statement 100
7.2.2. The `continue` Statement 101
7.3. BBC Thread: Computing the Greatest Common Divisor 103
7.4. AlgoRythmics Thread: Binary Search 105
7.4.1. Binary versus Linear Search 107
7.5. Lynx Thread: Primality Test 108
7.6. ECN Thread: Digital Clock 109
8. Pointers and One-Dimensional Dynamic Arrays 113
8.1. Pointers 113
8.1.1. Addresses of Variables 113
8.1.2. Declaring a Pointer 114
8.1.3. The `&` and `*` Operators 115
8.1.4. Multiple Pointers to the Same Variable 116

8.1.5. The NULL Pointer 116
8.2. The Relationship Between Pointers and One-Dimensional Arrays . . . 117
8.2.1. The Array Name and the Address of the First Element. 117
8.2.2. Indexing Through Pointers 118
8.2.3. Pointer Arithmetic in Arrays 119
8.2.4. How Does an Array Name Differ from a Pointer? 119
8.2.5. Traversing an Array with a Pointer 120
8.3. Dynamic Memory Allocation for One-Dimensional Arrays 121
8.3.1. The malloc and calloc Functions 122
8.3.2. Failed Memory Allocation 123
8.3.3. Reading and Printing a Dynamic Array 123
8.3.4. The free Function 125
8.4. AlgoRythmics Thread: Merging Two Sorted Sequences 126
8.5. Lynx Thread: In-Place Partitioning 129
8.6. BBC Thread: The Traveling Salesman and the Greedy Route 130
8.7. ECN Thread: Processing a Large Matrix Using Two Rows 134
9. Advanced Chapter – Connecting and Deepening 138
9.1. Learning More About Variables 138
9.1.1. Variable Types 139
9.2. Constants 142
9.2.1. Integer Constants 142
9.2.2. Floating-Point Constants. 143
9.2.3. Character Constants 143
9.2.4. String Constants 144
9.2.5. Symbolic Constants 144
9.2.6. Floating-Point Pitfalls 145
9.3. Format Specifiers in scanf and printf 146
9.3.1. Formatted Output – Field Width and Precision 148
9.4. Arrays 150
9.4.1. A Brief Overview of One-Dimensional Arrays. 150
9.4.2. Two-Dimensional Arrays 151
9.4.2.1. Processing Two-Dimensional Arrays 152
9.4.2.2. Symmetries in Square Matrices 153
9.4.3. Multidimensional Arrays (n-Dimensional Generalization). 155
9.5. Advanced Use of Pointers 156
9.5.1. Explicit Type Conversion of Pointers 156
9.5.2. The void * Pointer 158
9.5.3. const and Pointers 159
9.5.4. Two-Dimensional Arrays and Pointers. 161
9.5.5. Dynamically Created Two-Dimensional Arrays. 162
10. Advanced Chapter: Expressions and Control – The True Face of C 166
10.1. Expressions. 166

10.1.1. Operands and Operators 166
10.1.2. Operator Precedence and Parentheses 167
10.1.3. The Value and Type of Expressions 167
10.1.3.1. Implicit Type Conversion 168
10.1.3.2. Explicit Type Conversion 169
10.1.4. Arithmetic Operators 169
10.1.4.1. Increment and Decrement Operators 169
10.1.5. Relational and Logical Expressions 170
10.1.6. The Conditional Operator (?:) 171
10.1.7. The Comma Operator (,) 172
10.1.8. Assignment as an Expression 172
10.2. Bitwise Operations and Bit Masks 174
10.2.1. Bitwise Logical Operators 174
10.2.2. Bitwise Negation 175
10.2.3. Bit-Shift Operators 176
10.2.4. Bit Masks 177
10.2.5. Storing Multiple States in a Single Value 179
10.2.6. Compound Bitwise Assignments 180
10.2.7. Precedence and Parentheses 181
10.3. Expression Statements 183
10.3.1. The Essence of the Expression Statement: Effect, Not Value . . . 183
10.3.2. Which Expressions Are Suitable as Statements? 184
10.3.3. Function Calls as Expression Statements 184
10.3.4. Expression Statements and Control 185
10.3.5. The Danger of Overly Compact Expression Statements 185
10.3.6. Null and Compound Statements – Syntactic Constraint, Semantic Freedom 186
10.4. Value-Controlled Branching: The `switch` Statement 187
10.4.1. The `switch` Structure 187
10.4.2. "`switch-break`" and the Fall-Through Phenomenon 189
10.4.3. Limitations of `switch` 190
10.5. Iteration as a Control Model 191
10.5.1. Condition-Controlled Iteration 191
10.5.2. Event-Controlled Iteration 191
10.5.3. Menu-Driven Program: A Mini Calculator in Two Approaches . . 192
10.5.4. Multiple Levels of Nesting 194
11. Functions: Taking Subproblems into Our Own Hands 196
11.1. Function Definition, Function Declaration, and Function Call 197
11.2. Parameter Passing 202
11.3. Passing Arrays to Functions 206
11.3.1. One-Dimensional Arrays as Function Parameters 206
11.3.2. Passing Two-Dimensional Arrays to Functions 208

11.4. BBC Thread: Sudoku Checker 210
11.5. AlgoRythmics Thread: Bubble Sort 212
11.6. Lynx Thread: Beavers and Vipers 215
11.7. ECN Thread: Mountain Ridge in a Matrix 216
12. Recursion: When a Function Calls Itself 218
12.1. Recursion: An Alternative to Loops 218
12.2. The Outward and Return Journey of Recursion 220
12.3. Recursive Functions That Compute a Value 223
12.3.1. When a Recursive Formula Already Exists 223
12.3.2. Digit-by-Digit Processing 224
12.3.3. Element-by-Element Processing of a Number Sequence 225
12.4. AlgoRythmics Thread: Recursive Search and Sorting 227
12.4.1. Recursive Binary Search 227
12.4.2. Recursive Bubble Sort 228
12.4.3. When Recursion Is Natural: Merge Sort 229
12.5. BBC Thread: PageRank – When Importance Depends on the Importance of Others 231
12.6. Lynx Thread: Can the Lynx Escape the Maze? 233
12.7. ECN Thread: Dependencies – What Must Be Created First 235
13. Characters and Strings 239
13.1. Characters: When There Is a Number Behind a Letter 239
13.2. Strings: Text as a Character Array 241
13.2.1. Reading and Printing Strings 241
13.2.2. Operations on Strings 243
13.2.3. The `<string.h>` Library 245
13.3. Processing Text Files Character by Character, Word by Word, and Line by Line 247
13.4. AlgoRythmics Thread: Building a Name List with Insertion Sort . . . 250
13.5. BBC Thread: The Stable Marriage Problem – Names, Lists, and Matchings 254
13.6. Lynx Thread: Caesar Cipher 257
13.7. ECN Thread: Simplifying a String 258
14. Structures: When Multiple Data Items Belong to One Object 262
14.1. The First Structure 262
14.2. Reading and Printing Fields 263
14.3. A Structure with Fields of Different Types 264
14.4. Arrays of Structures 265
14.5. Passing a Structure to a Function 266
14.6. Structures and `typedef` 269
14.7. Keeping Related Data Together 270
14.8. AlgoRythmics Thread: Selection Sort for Structures 272
14.9. BBC Thread: Scheduling – When Order Matters 275

14.10. ECN Thread: Efficient Sorting and Searching with Library Functions . . . 279
14.11. Lynx Thread: Recording Marriages and Divorces . . . 282
15. File Handling: When Data Persists . . . 286
15.1. Opening and Closing Files . . . 286
15.2. Text Files . . . 287
15.3. Text and Binary Files . . . 289
15.4. Writing to a Binary File: `fwrite` . . . 290
15.5. Reading from a Binary File: `fread` . . . 291
15.6. Writing a Structure to a Binary File . . . 292
15.7. Saving and Reloading an Array of Structures . . . 292
15.8. Random Access in a Binary File . . . 295
15.9. AlgoRythmics Thread: Quicksort on Records Read from a Binary File . . . 296
15.10. BBC Thread: Face Recognition, Image Processing, and Binary Image Files . . . 300
15.11. Lynx Thread: Equilibrium Points in Wildlife-Camera Images . . . 303
15.12. ECN Thread: When Input and Output Are Part of the Problem . . . 305
16. Advanced Chapter: Advanced Tools, Deeper Layers . . . 310
16.1. Function Pointers: When We Pass an Operation . . . 310
16.2. Variadic Functions . . . 313
16.3. Parameters of `main` . . . 313
16.4. Indirect Recursion . . . 314
16.5. Strings: Lines, Fields, and Memory Areas . . . 315
16.6. Organizing Types: `typedef`, `enum`, `union` . . . 317
16.7. File Handling: Characters, Lines, and Positions . . . 319
16.8. The Role of the Preprocessor: `include`, `define`, and Macros . . . 320
17. C Programming from a Bird's-Eye View: The Threads Converge . . . 324
17.1. The C Thread: The Internal Order of a Program . . . 325
17.1.1. From Algorithm to Program . . . 325
17.1.2. The Program as a Structure . . . 326
17.1.3. The Big Picture of Types . . . 327
17.1.4. Global, Local, Static, and Dynamic Variables . . . 328
17.2. AlgoRythmics Thread: Algorithms in Motion . . . 331
17.3. BBC Thread: Algorithms Behind the Scenes of the World . . . 331
17.4. Lynx Thread: Sharp Thinking Before Coding . . . 332
17.5. ECN Thread: Algorithms Put to the Test . . . 333
References . . . 335
Abstract. Code in C! – The Fundamentals of Programming, Step by Step . . . 337
About the Author . . . 338

1. Bevezetés

A 21. század a számítás és az algoritmusok kora. A világ működését ma már láthatatlanul összefonják a számítási folyamatok: közösségi hálók, közlekedésirányítás, mesterséges intelligencia, pénzügyi tranzakciók és orvosi diagnosztika egyaránt algoritmusokra épül. Ezek az algoritmusok döntéseket hoznak, adatokat elemeznek, modelleket alkotnak – és egyre inkább meghatározzák, hogyan élünk és gondolkodunk.

A programozás ezért nem pusztán technikai ismeret, hanem a 21. század új írástudása (Wing 2006). Ugyanúgy, ahogy az írás-olvasás a 19–20. században a társadalmi fejlődés alapja lett, ma az algoritmikus gondolkodás és a kódolás tölti be ezt a szerepet. Aki programozni tanul, nemcsak egy gép nyelvét tanulja meg, hanem rendszert alkot, gondolkodni tanul, és megérti a logika és a kreativitás határtereit.

1.1. Az algoritmus mint gondolkodás és mint kultúra

A programozás mélyén az algoritmus áll – a logikus, lépésről lépésre megfogalmazott gondolatmenet, amelyet a gép és az ember egyaránt követhet. Az algoritmus tervezése és értelmezése strukturált gondolkodást igényel: az ember megtanul problémát lebontani, sorrendet, feltételt, ismétlődést felismerni, és azokat szabályos formába önteni.

A BBC dokumentumfilmje, *The Secret Rules of Modern Living: Algorithms*, érzékletesen mutatja be, hogy az algoritmusok milyen mélyen áthatják a társadalmat – és mennyire fontos, hogy ne csak használói, hanem értői is legyünk ezeknek (BBC 2015). A film rámutat: az útvonaltervezéstől a párosítási rendszereken, az orvosi diagnosztikán és a pénzügyi folyamatokon át egészen a közösségi média működéséig minden mögött számítási logika áll. Az algoritmus ismerete tehát felelősséget is jelent: minden szoftver mögött emberi döntések, értékek és hatások húzódnak meg. A programozás tanulása így egyszerre technikai és etikai fejlődési folyamat: logikát, figyelmet, előrelátást és felelősségérzetet fejleszt.

1.2. Miért C? – az alap, amelyre minden más épül

A mai világban számtalan modern, „felhasználóbarát” nyelv áll rendelkezésünkre – Python, Java, C#, JavaScript stb. Mégis, ha a cél az, hogy valaki valóban megértse a számítógép működését, a C nyelv az ideális kiindulópont.

A C nyelv közel van a géphez, de még emberi logikával követhető (Kernighan–Ritchie 1988). Megmutatja, hogyan épül fel a memória, hogyan kezel a gép adatokat, mi történik a háttérben minden egyes műveletnél. Ez a tudás különösen fontos a mérnökhallgatóknak, akiknek a hardver és szoftver közti kapcsolatot, az erőforrás-kezelést vagy a beágyazott rendszereket is érteniük kell.

A C nem takar el semmit. Értjük ez alatt, hogy sok modern nyelvnél kevesebb részletet rejt el a memória- és erőforrás-kezelésből: nincs „automatikus szemétgyűjtés", nincsenek „varázslatos absztrakciók". Minden döntés a programozóé – és ezzel együtt a felelősség is. A C nyelv tehát nem csupán eszköz, hanem iskola: megtanít pontosan, fegyelmezetten és logikusan gondolkodni.

1.3. A könyv többszálú felépítése

A *Kódolj C-vel!* könyv nem hagyományos tankönyv. Nem egyetlen vonalon vezeti végig az olvasót, hanem több szálon fonódó tanulási útvonalat kínál, amelynek mindegyik eleme más-más oldalról közelíti meg a programozást és az algoritmikus gondolkodást.

1.3.1. A C szál – a programozás szerkezeti alapjai



Ez a könyv fő szála. Itt ismerkedünk meg a programozás alapelemeivel – változókkal, ciklusokkal, függvényekkel, mutatókkal, struktúrákkal stb. A cél, hogy az olvasó lépésről lépésre építse fel a tudását, és közben megértse, mi történik a háttérben. A C szál a könyv gerince, amelyre minden más ráépül: a további szálak ezekre az alapokra reflektálnak, és azok alkalmazásait szemléltetik.

1.3.2. A Hiúz szál – az algoritmikus gondolkodás építése



A Bebras Challenge világméretű oktatási kezdeményezés, amely játékos, logikai és algoritmikus feladatokon keresztül fejleszti a tanulók problémamegoldó és informatikai gondolkodását (Bebras 2026). Magyarországon a program megvalósítását az Eötvös Loránd Tudományegyetem Informatikai Kara koordinálja HÓDítsd meg a biteket! – e-Hód néven (ELTE 2026). A hazai szervezők feladatai közé tartozik a nemzetközi versenyfeladatok magyar nyelvű adaptálása, valamint a verseny magyarországi megszervezése és lebonyolítása. Ezek a feladatok nem igényelnek programozási ismeretet, mégis kulcsfogalmakat – mint például az adatszervezés, a lépésről lépésre való gondolkodás vagy a hatékony döntéshozatal – ültetnek el a tanulók fejében.

Könyvünk azonban nem egyszerűen átvesz ezekből a feladatokból, hanem továbbgondolja és adaptálja őket a C programozás világához. Ennek jelképeként a Hód helyett a Hiúz szimbólumot használjuk: a hiúz éleslátása és figyelme azt a

tudatos, analitikus gondolkodást testesíti meg, amely a jó programozót jellemezi. A „Hiúz szál” így szervesen illeszkedik a könyv koncepciójába: e feladatok segítségével előbb felfedezzük a logikai szerkezeteket, majd lépésről lépésre megvalósítjuk őket C nyelven, hidat építve a gondolat és a kód között.

1.3.3. Az AlgoRythmics szál – táncoló algoritmusok



Az AlgoRythmics projekt az algoritmusok oktatásának és megértésének kreatív, interdiszciplináris megközelítését képviseli (AlgoRythmics 2026). A videósorozat azzal a céllal jött létre, hogy a programozás absztrakt logikáját érzékelhető, esztétikai élménnyé alakítsák. Az AlgoRythmics kollekció különleges megközelítést kínál: tánckoreográfiákon keresztül mutatja be az algoritmusok működését. A mozgás és a ritmus révén testet ölt a logika – az összehasonlítások, cserék, ciklusok és sorrendiségek vizuálisan is megjelennek. A különböző rendezési és keresési algoritmusokat néptáncokra komponált koreográfiákban jelenítik meg, ahol a tánclépések a program elemeit, az összehasonlításokat és cseréket szimbolizálják.

E könyv ezeknek a táncoknak az elemzését is tartalmazza, hogy az olvasó ne csak kód formájában, hanem mozgásban, „élő algoritmusként” is lássa, mit jelent például egy rendezés vagy keresés. A mozgás nyelvén így nemcsak az algoritmusok működése válik intuitívan érthetővé, hanem az is, hogyan szerveződik rend a káoszból – miként válik az algoritmikus gondolkodás egyfajta ritmikus, harmonikus folyamattá.

1.3.4. A BBC szál – algoritmusok a valóságban



A BBC dokumentumfilmje inspirálta a könyv azon részét, amely valós rendszerek algoritmusait tárgyalja (BBC 2015). A cél, hogy az olvasó lássa: az algoritmus nem elméleti fogalom, hanem valós hatású eszköz. E betétek konkrét példákon keresztül mutatják be, hogyan találkozik az elmélet a gyakorlattal, és miként alakítják az algoritmusok a modern világ döntéseit.

1.3.5. Az ECN szál – algoritmika váll váll mellett



A Sapientia ECN programozóversenye a régió egyik meghatározó informatikai megmérettetése, amely az ACM-ICPC nemzetközi formátumát követi (Sapientia EMTE 2026). Háromfős csapatok egyetlen számítógépen, angol nyelvű feladatokon dolgoznak öt órán keresztül, így a verseny egyszerre fejleszti a csapatmunkát, a logikus gondolkodást és az időnyomás alatti teljesítőképességet. A közepes nehézségű, de kihívást jelentő feladatok kizárólag a teljesen helyes és hatékony megoldásokat jutalmazzák, ezzel a verseny a precizitást, a stratégiai gondolkodást és a versenyszellemet állítja középpontba.

Versenyfeladatokból átdolgozott példák révén e könyv komplexebb, alkalmazott algoritmikai helyzetekbe vezeti be az olvasót. Ezekben az alfejezetekben már nemcsak az a cél, hogy egy program működjön, hanem hogy hatékony, elegáns és skálázható legyen. A versenyhangulat és a kreatív problémamegoldás segít abban, hogy a tanulás élménnyé váljon – nemcsak tananyaggá.

1.4. AI-cseppek



E könyv egyik további sajátossága az AI-cseppeknek nevezett betétek megjelenése, amelyek a gondolatvezetés mind az öt szálán felbukkannak. A mesterséges intelligencia (AI – *Artificial Intelligence*) ma már könnyen elérhető eszköz, amely képes kódot írni, magyarázatokat adni és alternatív megoldásokat javasolni. Ez azonban nem teszi feleslegessé a tanulást – épp ellenkezőleg: új felelősséget ró a tanulóra.

Az AI-csepp betétekben az AI nem megoldásként, hanem tanulási segédeszközként jelenik meg. Olyan helyzetekben vonjuk be, ahol kérdezésre ösztönöz, rávilágít rejtett feltételezésekre, segít pontosabban megfogalmazni egy gondolatot, vagy ellenőrző tükörként működik.

Az AI használata nem helyettesíti a feladat megértését, a kézi végiggondolást, a fordítást, a futtatást és az ellenőrzést. Fontos alapelv, hogy amit az AI állít, azt mindig ellenőrizni kell – a könyv szövegével, a fordítóval, a futó programmal és a józan ész segítségével. Az AI-cseppek célja nem az, hogy lerövidítsék az utat a válaszig, hanem az, hogy láthatóvá tegyék az utat, amely a válaszhoz vezet.

1.5. Kiemelések

A következő fejezetekben az egyes gondolati egységeket és kiemeléseket ikonokkal jelöljük, hogy segítsük az eligazodást és az anyag tudatos feldolgozását.

- **</> Szintaxis:** egy adott nyelvi elem általános alakjának, formai felépítésének és grammatikájának leírása.
- **[i] Megjegyzés:** olyan kiegészítő információk vagy pontosítások, amelyek segítik a megértést, de nem feltétlenül részei a fő gondolatmenetnek.
- ☞ **Tanulság:** olyan felismerések, amelyek az adott példán túlmutatva más feladatoknál is hasznosak lehetnek.

- ▦ **Algoritmikus mintázat:** visszatérő megoldási sémák, amelyek különböző problémákban azonos szerkezetben jelennek meg.
- ⏱ **Időbonyolultság:** annak vizsgálata, hogy egy algoritmus végrehajtásához szükséges műveletek száma miként függ a bemenet méretétől, különös

tekintettel a legjobb és legrosszabb esetre. Segítségével a különböző algoritmusok hatékonysága összehasonlítható, függetlenül a konkrét számítógéptől vagy futási környezettől.

- ◆ **Betét:** kiegészítő magyarázat, érdekesség vagy rövid kitérő, amely segíthet az anyag mélyebb megértésében, de kihagyása nem akadályozza a fejezet további követését.

- ▶ **Összegzés:** az addig tárgyalt fogalmak és megoldások rendezett, áttekinthető összefoglalása.
- ▶▶ **Zárógondolat:** az adott témát lezáró megállapítások, amelyek gyakran előre is mutatnak a következő fejezetek irányába.

Ezek a jelölések azt mutatják, mikor érdemes nem tovább olvasni, hanem elgondolkodni az addig tanultakon.

1.6. A Pareto-elv szellemében épülő könyv

A *Kódolj C-vel!* a Pareto-elv logikáját követi: nem törekszik arra, hogy azonnal mindent elmondjon, hanem először arra a kritikus 20%-ra koncentrál, amely a tudás 80%-át megalapozza (Koch 1997).

Az algoritmikus gondolkodás és a programozás elsajátítása során ez azt jelenti, hogy először a legfontosabb alapfogalmakra és mintákra építünk, amelyek a későbbi, bonyolultabb rendszerek megértéséhez is kulcsot adnak. A részletesebb technikai mélységeket, ritkább nyelvi szerkezeteket és optimalizálási trükköket a könyv „mesterfejezeteiben" tárgyaljuk – ezek az alapokra építve vezetik el az olvasót a magasabb szintű megértés felé, fokozatosan bővítve és mélyítve a megszerzett tudást.

Ezzel a megközelítéssel a könyv célja nem az, hogy információval terhelje az olvasót, hanem hogy strukturált tudást építsen, amely tartós, átgondolt, és új helyzetekben is alkalmazható.

1.7. Sapientia – ahol az utak összeérnek

A *Bebras Challenge*, a BBC dokumentumfilm, az AlgoRythmics tánckoreográfiák és az ECN programozóverseny különböző nézőpontokból közelítik meg ugyanazt a célt: az algoritmikus gondolkodás fejlesztését és a számítási kultúra terjesztését.

E kezdeményezések összefonódása e tankönyvben nem véletlen. Mindegyik valamilyen módon kapcsolódik a Sapientia Erdélyi Magyar Tudományegyetem Matematika–Informatika Tanszékén zajló programozásoktatáshoz: az AlgoRythmics projekt a Sapientia egyedi kutatási műhelyében született; a BBC filmje az

AlgoRythmics táncait használja a rendezési algoritmusok szemléltetésére; az ECN verseny több mint másfél évtizede a Sapientia szervezésében zajlik; a *Bebras Challenge* pedig az AlgoRythmicshez hasonlóan a számítógépes gondolkodás fejlesztését tűzte ki célul, és gyakran szolgál inspirációul számunkra.

Ezek a szálak nem csupán a könyv fő, C szálára fonódnak rá, hanem egymást is jól kiegészítik: a BBC a valóságot hozza közelebb, az AlgoRythmics a művészet eszközeivel teszi láthatóvá a logikát, a Hiúz-betétek a gondolkodás alapjait építik, az ECN pedig a versenyeztetés révén mélyíti el mindezt. Együtt olyan szellemi közösséget alkotnak, ahol az algoritmus nemcsak tananyag, hanem gondolkodásmód, kreatív kifejezés és közös nyelv.

2. Algoritmikus gondolkodás és programozás

Mielőtt konkrét programokat írnánk, érdemes megérteni, hogyan „gondolkodik” a számítógép, és milyen szemléletre van szükség ahhoz, hogy hatékonyan tudjuk utasítani. A programozás nem pusztán nyelvi szabályok megtanulása, hanem problémák világos, lépésekre bontott megfogalmazása a gép számára.

Ebben a rövid felvezető fejezetben azt vizsgáljuk meg, miként alakul át az emberi gondolat algoritmussá, majd végül futtatható programmá. Megismerkedünk az algoritmikus gondolkodás alapelveivel, a bemenet–feldolgozás–kimenet modelljével, valamint azzal a szereppel, amelyet a programozási nyelvek és a fordítóprogramok töltenek be ebben a folyamatban.

2.1. Hogyan érteti meg magát az ember a géppel

A számítógép az ember egyik legpontosabb, de legszűkszavúbb eszköze. Soha nem fárad el, nem hibázik – de csak akkor, ha pontosan értelmezi, amit kérünk tőle. És ez a nehézség: nem a gép gondolkodik, hanem mi próbáljuk úgy programozni, mintha képes lenne gondolkodni.

Az ember természetes nyelve – mint a magyar vagy az angol – rugalmas és sokértelmű. A gép viszont csak 0-t és 1-et ismer: minden adat, parancs és döntés végül bitek (0/1) sorozatává válik. Ez a gép anyanyelve, a gépi kód – amit ő tökéletesen ért, mi viszont alig. Ezért születtek meg a programozási nyelvek: olyan mesterséges nyelvek, amelyek az emberi gondolatot a gép számára „érthető” formába öntik. A fordítóprogram (compiler) ennek a tolmácsa: elolvassa az emberi nyelvhez közel álló utasításokat, és gépi kódra fordítja őket, amit a processzor már végre tud hajtani.

2.2. A C nyelv – a programozás közös gyökere

A C nyelv az 1970-es évek elején született (Ritchie 1993), és rövid idő alatt az egész informatikai világ közös „gyökérnyelvévé” vált. Számos széles körben használt nyelv szintaxisa és vezérlési szerkezete közvetlenül vagy közvetve a C hagyományához kapcsolódik.

A C nyelv egyszerre emberközeli és gépközeli. Képes leereszkedni a hardver szintjére, mégis elég magas szintű ahhoz, hogy tisztán és logikusan fejezzük ki a gondolatainkat. Ezért nevezik a C-t gyakran a „legmagasabb alacsony szintű

nyelvnek”. Nem véletlen, hogy a programozást tanulók számára a C az egyik legjobb kiindulópont: tanít pontosan gondolkodni, érteni a gépet, és szabatosan megfogalmazni a műveletsorokat.

A C nyelv sikerét az is mutatja, hogy számos rendszer és fordító fejlesztette tovább (ISO/IEC 2024). Hogy ne alakuljon ki káosz, az évek során szabványosították:

- C89/C90 – az első hivatalos ANSI/ISO szabvány
- C99 – bővítések, kényelmi funkciók
- C11, C17, C23 – fokozatos modernizálás

Ezek a változatok mind ugyanazt a „nyelvi magot” őrzik. Ebben a könyvben is hangsúlyosan erre a stabil, időtálló C-magra építünk: arra a tudásra, amely minden C nyelvű programozás alapja.

2.3. Algoritmus és kód

A programozás lényege az, hogy feladatokat oldjunk meg a számítógép segítségével. De mielőtt a gép megoldhatná a problémát, nekünk kell kitalálnunk, hogy pontosan mit és hogyan tegyen. A gép nem gondolkodik helyettünk – csak végrehajtja azt, amit világosan, lépésről lépésre előírtunk neki. A programozás tehát nem csupán utasítások leírása, hanem gondolkodás a „gép logikájával”.

A valódi kihívás az, hogy leereszkedjünk a gép szintjére, és az ő korlátozott, de pontos eszközkészletével gondolkodjunk. Minden műveletet, amit az ember egyetlen pillanat alatt átlátna, a gép számára kis, precízen meghatározott lépésekre kell bontani. A programozó olyan, mint egy karmester, aki a primitív gép számára megírja a partitúrát: megmondja, mit játsszon, mikor és milyen sorrendben, hogy a végén a kívánt eredmény „megszólaljon”.

Fontos különbséget tenni algoritmus és kód között. Az algoritmus a megoldás logikus, ember által megfogalmazott terve – egy olyan lépéssorozat, amely a gép gondolkodási korlátaihoz igazodik, de még nem kötődik konkrét programnyelvhez. A kód e terv konkrét megvalósítása: az algoritmus leírása egy adott programozási nyelv, például a C utasításaival.

Másképp fogalmazva: az algoritmus a gondolat, a kód a megvalósítás. Az algoritmus „kitalálja”, mit kell tenni, a kód pedig „megmondja”, hogyan. A kód utasítja a számítógépet, hogy hajtsa végre az algoritmus által meghatározott műveletsort – így lesz a gondolatból működő rendszer, a tervből futó program.

2.4. A B–F–K gondolkodás

Minden algoritmus – legyen az egyszerű vagy sokszorosan összetett – ugyanarra az alaplogikára épül: Bemenet → Feldolgozás → Kimenet, röviden B–F–K. Ez a háromlépéses minta nemcsak a számítógépes működés váza, hanem a

problémamegoldás egyetemes modellje is. Az a gondolkodási ritmus, amelyben az ember és a gép találkozik.

- Bemenet (Input): az információ belépése a rendszerbe. A bemenet az a pillanat, amikor az algoritmus kapcsolatba lép a külvilággal. Itt gyűjtjük be azokat az adatokat, amelyekre a feldolgozás épülni fog – ez kezdetben billentyűzetről beírt vagy fájlból beolvasott számadatokat fog jelenteni.
- Feldolgozás (Processing): a logika tere. A feldolgozás az algoritmus „gondolkodási" szakasza, ahol a bemeneti adatokból értelmes eredmény születik. Itt zajlanak a számítások, döntések, összehasonlítások és ismétlések – minden, ami a feladat megoldásához vezet. A feldolgozás maga az algoritmus lényegi része, ahol a nyers adatok átalakulnak struktúrává, jelentéssé, válasszá. Ebben a szakaszban építi be a programozó az értelmet a folyamatba.
- Kimenet (Output): az eredmény megjelenése. A kimenet az algoritmus látható eredménye – a folyamat zárópontja, amikor az információ visszatér a külvilágba. Ez egyelőre képernyőn megjelenő üzenet vagy fájlba írt adatok formájában fog megvalósulni, de történhet hangként, animációként vagy akár egy robotkar mozdulataként is. A kimenet az, amit az ember érzékel: a gép válasza a világra. Egy jó algoritmus esetében ez az eredmény nemcsak helyes, hanem hasznos és érthető is – a gondolat és a számítás közös műve.

Ez az egyszerűnek tűnő trió – Bemenet, Feldolgozás, Kimenet – az algoritmikus gondolkodás alaptérképe. Ahogy utaltunk már rá, minden bonyolultabb rendszer is erre a logikai vázra épül. Ha ezt a mintát megértjük, már nemcsak kódot írunk, hanem gondolkodásunkat is algoritmizáljuk: képesek leszünk világosan látni, mit adunk be, mit csinálunk vele, és mit kapunk vissza.

►► Zárógondolat

Az algoritmikus gondolkodás elsajátítása azt jelenti, hogy megtanulunk a számítógéphez igazodva gondolkodni: problémákat világosan megfogalmazni, lépésekre bontani és következetesen végrehajtható formába önteni. A programozási nyelv ebben nem cél, hanem eszköz. A következő fejezetekben ezt a szemléletet alkalmazva térünk át a konkrét C nyelvi szerkezetekre, ahol az elméleti gondolatmenetek kézzelfogható programokká válnak.

3. Első lépések: változók, értékadás és I/O-műveletek

Ebben a fejezetben megkezdjük az ismerkedést a C nyelvű programozás alapjaival. Elsősorban arra figyelünk, hogyan alakíthatunk át egy egyszerű feladatot a számítógép számára végrehajtható, pontos utasítássorrá. Közben megtapasztaljuk, hogy már a legegyszerűbb programok elkészítéséhez is világosan meg kell határoznunk, milyen adatokkal dolgozunk, milyen műveleteket végzünk rajtuk, és milyen eredményt várunk.

Az AlgoRythmics, BBC, Hiúz és ECN szál példáin keresztül azt is felismerjük, hogy a programozás jóval többet jelent a C nyelv utasításainak megtanulásánál. Gyakoroljuk a problémák részekre bontását, a megoldási lépések megtervezését, valamint a hibák felismerését és javítását. Első működő programjaink egyszerűek lesznek, mégis fontos mérföldkövet jelentenek: ezekkel tesszük meg első valódi lépéseinket az algoritmikus gondolkodás és a programozás világában.



E fejezet – C szálon – bemutatja a változók fogalmát, a standard bemenetről (billentyűzet/fájl) történő adatbeolvasás és a standard kimenetre (monitor/fájl) való eredménykiírás módját, valamint az aritmetikai műveleteket megvalósító operátorokat.

3.1. Változók – az adatok tárolói a memóriában

A számítógépes feladatmegoldás adattárolást igényel, amely a számítógép memóriájában, változók segítségével valósul meg. A memória bájtokra, vagyis 8 bites rekeszekre oszlik. A változó tulajdonképpen egy névvel ellátott tárolóhely e rekeszek között, amelyben a program ideiglenesen adatot helyezhet el. A változó elnevezés arra utal, hogy olyan tárolóról van szó, amelynek tartalma a program végrehajtása során megváltozhat. Ezzel szemben a konstansok (lásd később) értéke állandó marad.

A változók négy alapjellemzője az alábbiak:

1. *típus* – milyen jellegű adatot tartalmaz;
2. *azonosító (név)* – így hivatkozunk rá;
3. *cím* – a memória azon helye, ahol található;
4. *érték* – a ténylegesen tárolt adat.

A változó deklarálásával jelezzük a számítógépnek, hogy hozzon létre egy adott típusú tárolóhelyet. Másképp fogalmazva: amikor deklarálunk egy változót,

utasítjuk a számítógépet, hogy foglaljon le számára egy megfelelő memóriaterületet, és rendelje hozzá a megadott nevet.

</> Szintaxis: a változó deklarálás általános alakja

```
<típus> <azonosító>;
```

Egyelőre csak az `int` és `float` típusú változókkal ismerkedünk meg; előbbiek egész, utóbbiak valós számok tárolására szolgálnak. Az `int` kulcsszó az *integer* szóból ered, amely magyarul 'egész'-t jelent. A `float` elnevezés pedig a valós számok memóriabeli ábrázolásának módjára, a lebegőpontos (*floating-point*) reprezentációra utal.

Íme néhány példa változók deklarálására:

```
int kor;
float atmero;
int a, b, c;
float x, y, z;
int ev = 2025;
float pi = 3.14;
```

A példákból az is kiderül, hogy több azonos típusú változót egyetlen deklarálással is megadhatunk, és hogy a változók már deklaráláskor kaphatnak kezdőértéket. Ha nem adunk meg kezdőértéket, az értéke meghatározatlan (úgynevezett „memóriaszemét"). Ezért felhasználása inicializálás előtt hibás programviselkedéshez vezethet.

A változó típusa meghatározza annak memóriaméretét is: az `int` és a `float` változók jellemzően `4` bájtot foglalnak el, vagyis négy memóriacímnyi rekesz kerül fenntartásra számukra (ami `4 × 8 = 32` bitet tesz ki). Egy változó vagy típus memóriaméretét a `sizeof` segítségével kérdezhetjük le. Például a `sizeof(int)` értéke `4`.

Az utolsó két példában az `ev` és `pi` változók kezdőértékei tízes számrendszerben vannak megadva. Mivel a változók a memóriában jönnek létre, ahol minden adat `0` és `1` jelek (bitek) formájában kerül eltárolásra, a háttérben szükségszerűen megtörténik a tízes számrendszerű értékek bináris (`0`/`1`) alakra való konvertálása. Egyelőre fogadjuk el, hogy a `2025` egész szám `32` bites '*int* módú' bináris formává, míg a `3.14` valós szám 32 bites '*float* módú' bináris formává alakul át. Az egész számok fixpontos és a valós számok lebegőpontos belső ábrázolásának részletei végett lásd a (Kátai 2008) jegyzet függelékét.

A típusméretből adódik az az értékkészletet is, amelyből az adott típus érvényes értéket képes tárolni. Például egy `32` bites `int` típusú változóban az `−2147483648` és `2147483647` közötti intervallumból tárolható egy egész érték.

Ezek a látszólag titokzatos számok majd az első mesterfejezetben, a 9. fejezetben nyernek értelmet.

Érdemes beszédes változóneveket választani (például: `kor`, `atmero`, `ev`, `pi`). Az azonosítókra vonatkozó szabályok az alábbiak:

- csak betű, szám és aláhúzás (_) engedélyezett,
- nem kezdődhet számmal,
- kis- és nagybetű különbözik,
- nem lehet kulcsszó (`int`, `for` stb.).

Példák helytelen változónévre (//-el megjegyzést írhatunk egy kódsorhoz):

```
int 2szam;          // hibás, mert számjeggyel kezdődik
int float;         // hibás, mert a float kulcsszó
float valos szam;  // hibás, mert szóközt tartalmaz
```

Összegzésként: az adatok ideiglenes tárolására memóriaváltozókat használunk. A változók deklarálásakor meghatározzuk a típusukat, és megadjuk a nevüket. A típus határozza meg a változó által lefoglalt memória méretét, valamint azt az értékkészletet, amelyből az adott típus érvényes értéket képes tárolni.

A tényleges memóriafoglalást a rendszer végzi, ezért a változó memóriacímét úgymond a gép határozza meg. Amennyiben erre kíváncsiak vagyunk, a változó címe a címképző (`&`) operátor segítségével kérdezhető le: `&`<változónév>.

[i] Megjegyzés. A mai személyi számítógépek többségén egy bájt 8 bites, az `int` és a `float` pedig gyakran 4 bájtot foglal. Ezek azonban nem minden számítógépen szükségszerűen ilyen méretűek. Egy típus tényleges memóriaigényét, ahogy már említettük is, a `sizeof` operátor segítségével kérdezhetjük le.

3.2. Értékadás és aritmetikai operátorok

A fenti példákban már megjelent az értékadás operátor (=), amikor deklarálásakor kezdőértéket adtunk egy változónak. Az értékadás azonban természetesen nem korlátozódik a deklaráció során történő inicializálásra. Általánosan az értékadás művelete azt jelenti, hogy a bal oldalon szereplő változó felveszi a jobb oldalon álló kifejezés értékét.

</> Szintaxis: az értékadás általános alakja

```
<változó> = <kifejezés>;
```

Íme, néhány példa is:

```
int a, b, x = 14, y = 15;
float atlag;
a = 3;
b = a + 2;
atlag = (x + y) / 2.0;
```

Fontos megérteni, hogy az = operátor nem egyenlőséget vizsgál, hanem értékadást hajt végre. A bal oldalon mindig egy tárolóhely (változó) áll, míg a jobb oldalon egy kifejezés, amelynek az értéke kiszámításra kerül, majd eltárolódik a bal oldali változóban.

A kifejezésekben szerepelhetnek a matematikából ismert alapvető aritmetikai műveleteket megvalósító operátorok, és ezek végrehajtási sorrendje itt is szabályozható kerek zárójelekkel. Az 3.1. táblázat szemlélteti, hogy az összeadás, kivonás és osztás ugyanazzal az operátorral történik, mint a matematikában, míg a szorzást a `*` műveleti jel jelöli.

3.1. táblázat. *A C nyelv aritmetikai operátorai*

Operátor	Jelentés	Példa	Eredmény
+	összeadás	3 + 2	5
-	kivonás	7 - 4	3
*	szorzás	2 * 5	10
/	egész osztás	7 / 2	3
%	osztási maradék	7 % 3	1
/	valós osztás	7 / 2.0	3.5

Az osztás operátor (/) használatakor különös figyelmet kell fordítani az operandusok típusára. Amennyiben mind az osztandó, mind az osztó egész típusú, egész osztás történik, vagyis az eredmény az osztási hányados lesz. Ebben az esetben az osztási maradék a `%` operátor segítségével kérdezhető le.

Ha azt szeretnénk, hogy a `/` operátor valós osztást végezzen, akkor legalább az egyik operandusnak valós típusúnak kell lennie; ekkor az eredmény is valós lesz. Ezért szerepel a fenti példában az átlagszámítás nevezőjében a 2.0, vagyis a 2-es konstans valós értékként. Úgy is fogalmazhatunk, hogy egész típusnál (`int`) az osztás (`/`) „levágja” a tört részt, míg lebegőpontosnál (`float`) a valós értéket adja.

Ha az osztás valamelyik operandusa konstans, akkor kézenfekvő, hogy azt valós értékként adjuk meg (lásd fentebb). De mi a helyzet akkor, ha például mindkét operandus egész típusú változó, és mégis valós osztást szeretnénk végrehajtani közöttük? Ilyenkor a kifejezés kiértékelésének idejére rákényszeríthetjük az egyik operandusra a valós típust úgy, hogy elé, zárójelben, expliciten kiírjuk a (`float`) típuskonverziót.

```
int szamlalo, nevezo, egesz_eredmeny1, egesz_eredmeny2;
float valos_eredmeny1, valos_eredmeny2;
szamlalo = 13;
nevezo = 4;
egesz_eredmeny1 = szamlalo / nevezo;            // 3
valos_eredmeny1 = (float)szamlalo / nevezo;     // 3.25
egesz_eredmeny2 = (float)szamlalo / nevezo;     // 3
valos_eredmeny2 = szamlalo / nevezo;            // 3.0
```

Érdemes megemlíteni azt is, hogy a háttérben számos implicit konverzió történhet. Ha egy aritmetikai operátor egyik operandusa egész, a másik pedig valós típusú, akkor az egész típusú operandus a művelet végrehajtásának idejére automatikusan valós típusúvá alakul, és az eredmény természetesen valós lesz.

Értékadáskor a jobb oldali kifejezés típusát a bal oldalon szereplő változó típusához kell igazítani. Ez azt jelenti, hogy ha a jobb oldal valós típusú, akkor a bal oldali egész típusú változóban annak egész része kerül eltárolásra. A fenti példákban megjegyzésként megadtuk, hogy milyen értékek kerülnek eltárolásra a bal oldali változókban a jobb oldali osztások nyomán. Az utolsó két példa elsőre félrevezetőnek tűnhet, megértésük azonban tanulságos. A kifejezésekre és ezek kiértékelésének rejtelmeire egy későbbi fejezetben még visszatérünk.

Egy gyakori feladat, amikor egy változó aktuális értékét szeretnénk módosítani úgy, hogy ahhoz hozzáadunk egy értéket, abból kivonunk, megszorozzuk, elosztjuk, vagy éppen maradékot képezünk. Ezekre a klasszikus megoldások a következők:

```
a = a + 1;          // a-hoz hozzáad 1-et
b = b - 2;          // kivon a b értékéből 2-t
c = c * b;          // c-t megszorozza a b értékével
d = d / (b + c);    // d-t elosztja (b + c)-vel
e = e % 2;          // e felülíródik (e % 2)-vel
```

A C nyelv azonban lehetőséget ad ezeknek a műveleteknek egy tömörebb és kifejezőbb írásmódjára az úgynevezett összetett értékadó operátorok segítségével. Ezek a `+=`, `-=`, `*=`, `/=` és `%=` operátorok. A fenti példák ezekkel átírva:

```
a += 1;
b -= 2;
c *= b;
d /= (b + c);
e %= 2;
```

Azokban az esetekben, amikor egy változó értékét pontosan 1-gyel szeretnénk növelni vagy csökkenteni, használhatók az inkrementáló (++) és dekrementáló (--) operátorok:

```
++i;   // ekvivalens: i++; vagy i += 1; vagy i = i + 1;
--j;   // ekvivalens: j--; vagy j -= 1; vagy j = j - 1;
```

Ezek az operátorok használhatók prefix és posztfix formában is, vagyis a változó elé (`++i`, `--j`) vagy mögé (`i++, j--`) írva. A két forma közötti különbség csak akkor válik jelentőssé, ha az értéket egy összetettebb kifejezés részeként használjuk; erre egy később fejezetben térünk vissza.

3.3. Standard be- és kimenet

Már megkezdtük annak bemutatását, miként végezhetünk a számítógéppel aritmetikai műveleteket, és ezek eredményeit hogyan tárolhatjuk változókban. A legtöbb feladat esetében azonban ezt megelőzően szükség van a bemeneti adatok beolvasására, például a billentyűzetről, hogy a program ezeken végezhesse el a szükséges műveleteket. A programok többsége ugyanis nem önmagában működik, hanem külső adatokra támaszkodik. Másfelől, ugyanilyen fontos az is, hogy a számítások eredményeit miként jeleníthetjük meg kimeneti adatokként, például a képernyőn.

A C nyelv erre a célra a standard bemeneti és kimeneti csatornákat, valamint az ezek kezelésére szolgáló függvényeket biztosítja. A `scanf` függvény a standard bemenetről (általában a billentyűzetről) olvas adatokat, és azokat a megfelelő változókban tárolja el. A `printf` függvény ezzel szemben a standard kimenetre (általában a képernyőre) írja ki a szövegeket és a kiszámított értékeket.

3.3.1. A `scanf` függvény

A `scanf` első paramétere az idézőjelek közé írt formátumsor, amely leírja, hogy milyen típusú adatokat várunk a bemeneten. Ezt követik azoknak a változóknak a címei, amelyekbe az értékeket el szeretnénk helyezni. Hangsúlyozzuk, hogy a változók címeit kell megadni, ezért a legtöbb esetben az `&` (címképző) operátort kell használni.

Például egy egész szám beolvasása a következőképpen történhet:

```
int x;
scanf("%i", &x);
```

A % karakterrel bevezetett i formátumjel egy egész szám beolvasását jelzi, amelyet a scanf az x változóban tárol el. Amennyiben a billentyűzetről a 2025 karaktersort gépelnénk be (majd ENTER-t ütnénk), az x változóba a 2025 tízes számrendszerbéli egész *int*-módú bináris ábrázolása kerülne eltárolásra. E konverziót írja elő az i formátumjel.

Több adat beolvasására is lehetőség van egyetlen scanf hívással. Az alábbi példában a scanf a billentyűzetről begépelt, szóközzel vagy sortöréssel elválasztott két valós értéket olvas be.

```
float a, b;
scanf("%f %f", &a, &b);
```

Az i és f formátumjelek azt határozzák meg, hogy a beolvasott értékeket a scanf milyen módon alakítsa át: rendre int, illetve float típusú bináris ábrázolásra, mivel az adatok ilyen típusú változókban kerülnek eltárolásra. A scanf használatakor ezért különös figyelmet kell fordítani arra, hogy a megadott formátumjelek és a változók típusai összhangban legyenek, ellenkező esetben a program hibásan működhet.

A függvény visszatérési értéke megadja, hogy hány adat beolvasása történt meg sikeresen; ez hasznos segítséget nyújthat a hibák felismerésében (lásd később).

3.3.2. A printf függvény

A printf működése több szempontból hasonlít a scanf függvényhez: itt is egy idézőjelek közé írt formátumsorral kezdődik a függvényhívás. Ez esetben a formátumsor azt határozza meg, hogy egy kiírandó szövegben hol és milyen formában jelenjenek meg az értékek. E formátumsorban is % karakterrel bevezetett formátumjelek jelölik a kiírandó értékek helyét és típusát.

Mivel gyakran nem csupán az értékeket, hanem azok jelentését is meg szeretnénk jeleníteni, ezért a printf formátumsora rendszerint kísérő szöveget is tartalmaz (erre utaltunk feljebb úgy, mint 'kiírandó szöveg'), amely segíti a kimenet értelmezését. Az alábbi példában, a második printf esetében a kimeneten nemcsak az érték jelenik meg, hanem az is egyértelművé válik, hogy mit jelöl. A \n vezérlőkarakter sortörést ír elő, vagyis új sorba ugratja a kurzort, így a következő kiírás az előző alatt jelenik meg a képernyőn.

```
int x = 10;
printf("%i", x);                    // 10
printf("\nAz x erteke: %i\n", x);    // Az x értéke: 10
```

A printf függvényt gyakran használjuk a scanf kíséretében, különösen akkor, amikor a program a felhasználóval lép kapcsolatba. A printf ilyenkor tipikusan

a bemeneti adat megadására (a billentyűzetről) szólítja fel a felhasználót, míg a `scanf` a megadott adatot olvassa be. Például:

```
int x;
printf("Adjon meg egy egesz szamot: ");
scanf("%i", &x);
```

Ebben az esetben a `printf` nélkül a felhasználó nem kapna útmutatást arra vonatkozóan, hogy milyen adatot kell megadnia, ezért a két függvény együttes használata indokolt és szükséges, amennyiben felhasználóbarát kódot szeretnénk írni.

A `printf` formátumjelei típusonként határozzák meg, hogy az adott értéket milyen szöveges formában kell megjeleníteni. A leggyakrabban használt formátumjelek megegyeznek a `scanf` esetében alkalmazottakkal, például `%i` egész számokkal és `%f` valós számokkal. Ebben az esetben azonban a formátumjelek a fordított irányú konverziót írják elő: a belső, bináris ábrázolásból a tízes számrendszerbéli szöveges alakba történő átalakítást.

Bár hasznos tudni róluk, érdemes megjegyezni, hogy ezek az elsőre bonyolultnak tűnő konverziók automatikusan zajlanak a háttérben; a programozó feladata mindössze annyi, hogy a megfelelő formátumjeleket használja, valahányszor a `scanf` és a `printf` függvényeket alkalmazza.

3.3.3. A `freopen` függvény

Bizonyos esetekben előfordulhat, hogy a bemeneti adatokat nem közvetlenül a billentyűzetről szeretnénk beolvasni, vagy a kimenetet nem a képernyőn kívánjuk megjeleníteni. A `freopen` függvény segítségével lehetőség van a standard bemenet vagy a standard kimenet állományhoz történő átirányítására.

A standard bemenet átirányítását követően a `scanf` már nem a billentyűzetről várja a beolvasandó adatokat, hanem a megadott állományból olvas. Hasonló módon, a standard kimenet átirányítását követően a `printf` már nem a képernyőre írja ki az eredményeket, hanem a megadott állományba.

A standard bemenetre a `stdin`, a standard kimenetre pedig a `stdout` azonosítókkal hivatkozunk. Az alábbi példákban ezeket a `be.txt` és `ki.txt` állományokhoz irányítjuk át:

```
freopen("be.txt", "r", stdin);
freopen("ki.txt", "w", stdout);
```

Természetesen az első `freopen` feltételezi a `be.txt` állomány létezését: ezt előre létrehoztuk, és beleírtuk a beolvasandó bemeneti adatokat. A `ki.txt` állományt a program fogja létrehozni.

Az `r` (*read*) és `w` (*write*) paraméterek azt jelzik, hogy a `be.txt` olvasásra, míg a `ki.txt` írásra kerül megnyitásra. A függvény neve is erre utal: a `freopen` a standard adatfolyamokat egy másik állományhoz „újranyitja”.

Ha a standard bemenetet és kimenetet később vissza szeretnénk irányítani a billentyűzetre, illetve a monitorra, akkor ezt szintén a `freopen` függvény segítségével tehetjük meg.

```
freopen("CON", "r", stdin);
freopen("CON", "w", stdout);
```

Mivel a billentyűzet és a monitor együtt alkotják az úgynevezett konzolt (angolul *console*), amelyen keresztül a felhasználó alapértelmezetten kapcsolatba lép a számítógéppel, ezért az első `freopen` hívásban a CON a billentyűzetet (a konzol bemeneti egységét), míg a másodikban a monitort (a konzol kimeneti egységét) jelenti. Ezzel a standard bemenet és kimenet ismét a billentyűzethez és a képernyőhöz kerül visszairányításra.

3.4. Első C programjaink

Az elhíresült első C program az, amely mindössze annyit tesz, hogy a standard kimenetre, azaz a képernyőre kiírja a Hello world üzenetet.

```
#include <stdio.h>
int main(){
    printf("Hello world\n");
    return 0;
}
```

Ebből az első példából kiderül, hogy a számítógépnek szánt utasításainkat egy `main` nevű függvény keretében kell megadnunk, amelyet a `return 0;` utasítás zár le. Ezt egyelőre kezeljük egyfajta keretként, amelybe az utasításainkat elhelyezzük. Később visszatérünk erre, részletesen elemezzük, és értelmezzük a szerepét.

Egy másik fontos tanulság ebből az első programból az, hogy a bemeneti és kimeneti műveleteket lebonyolító könyvtári függvények – mint például a `scanf`, a `printf` vagy a `freopen` – használata feltételezi a `#include <stdio.h>` programsor jelenlétét. Ennek jelentését szintén egy későbbi fejezetben tárgyaljuk részletesen. Egyelőre tekintsük ezt is a program alapvető keretéhez (félkövér karakterekkel írtuk) tartozó elemnek.

Az alábbi program a fejezetben tanultak alkalmazásával két egész számot olvas be a billentyűzetről, majd a képernyőre kiírja az azokon végrehajtott alapvető aritmetikai műveletek eredményeit. (A `//` jelöléshez hasonlóan a `/*` és `*/`

„zárójelek” is megjegyzések írására szolgálnak, azzal a különbséggel, hogy ezek több sorra kiterjedő kommentek is lehetnek.)

```
#include <stdio.h>

int main(){

    int x, y;

    /* Bemenet: felhasználó tájékoztatása */
    printf("Adjon meg két egész számot: ");

    /* Adatok beolvasása */
    scanf("%i %i", &x, &y);


    /* Eredmények kiszámítása és megjelenítése */
    printf("Beolvasott adatok: x = %i, y = %i\n", x, y);
    printf("Osszeg: %i\n", x + y);
    printf("Kulonbseg: %i\n", x - y);
    printf("Szorzat: %i\n", x * y);
    printf("Egesz osztas: %i\n", x / y);
    printf("Maradek: %i\n", x % y);
    printf("Valos osztas: %f\n", x / (float)y);

    return 0;
}
```

Az alábbi változatban a beolvasás a `kezdoertekek.txt` állományból, míg a kiírás a `eredmeny.txt` állományba történik. A feladat abban áll, hogy beolvassuk a kezdőpontszámot és a bónuszértéket, majd az értékeket különböző összetett értékadó, valamint inkrementáló és dekrementáló operátorok segítségével módosítsuk, végül a kapott eredményt fájlba írjuk ki.

```
#include <stdio.h>

int main(){

    freopen("kezdoertekek.txt", "r", stdin);
    freopen("eredmeny.txt", "w", stdout);

    /*
       A kezdoertekek.txt-be például ezt írhatjuk:
```

```
       0 5
    */
    int pont, bonusz;
    scanf("%i %i", &pont, &bonusz);

    pont += 10;         // játékos pontot szerez
    pont++;             // újabb sikeres lépés

    pont -= 3;          // büntetés
    pont--;             // egy hiba

    pont *= 2;          // dupla pontszám
    pont += bonusz;     // bonusz hozzáadása

    printf("Vegso pontszam: %i\n", pont);

    /* Visszairányítás konzolra */
    freopen("CON", "r", stdin);
    freopen("CON", "w", stdout);

    /* Most már újra a képernyőre írunk */
    printf("Az eredmeny a eredmeny.txt fajlban van.\n");

    return 0;
}
```

Gyakorló feladatok

- Három egész szám átlaga: írj programot, amely három egész számot olvas be a billentyűzetről, majd kiszámítja és kiírja azok átlagát!
- Mértékegység-átváltás (m/s → km/h): készíts programot, amely egy sebességértéket olvas be méter per másodpercben, majd azt kilométer per órára váltja át, és az eredményt a képernyőn megjeleníti!
- Téglatest térfogata: írj programot, amely beolvassa egy téglatest három élének hosszát, majd kiszámítja és kiírja a test térfogatát!
- További példák végett lásd a (Kátai 2008) jegyzet 2.7. alfejezetét.

▶ **Összegzés.** Ebben a fejezetben, C szálon, kóstolót kaptunk a változók deklarálásáról és használatáról, az értékadás alapvető formáiról, valamint a C nyelv bemeneti és kimeneti műveleteiről. Lezárásként foglaljuk össze ezeknek a szerkezeteknek az általános alakjait.

- Változó deklarálása: `<típus> <azonosító>;`
- Változó deklarálása és inicializálása: `<típus> <azonosító> = <érték>;`
- Értékadás: `<azonosító> = <kifejezés>;`
- Bemenet: `scanf("<formátumsor>", <változócím_lista>);`

- Kimenet: `printf("<formátumsor>", <kifejezés_lista>);`
- Bemenet átirányítása: `freopen("<fájlnév>", "<mód>", stdin);`
- Kimenet átirányítása: `freopen("<fájlnév>", "<mód>", stdout);`

A következő fejezetekben ezekre a C nyelvi alapokra építve ismerkedünk meg az összetettebb szerkezetekkel.



AI-csepp: tipikus hibák

Kérdezd meg az AI-t: melyek a leggyakoribb hibák, amelyeket a kezdők elkövetnek a változók, értékadás, aritmetikai operátorok, `scanf` és `printf` használatakor C-ben?

- A válasz elolvasása után jelöld meg azokat a hibákat, amelyek a jelen fejezet példáiban is előfordulhatnának.

Kérdezd meg az AI-t: mi történik pontosan a memóriában, a háttérben, a következő sor kiértékelésekor?

```
valos_eredmeny2 = szamlalo / nevezo;
```

- A válasz elolvasása után ellenőrizd, hogy összhangban van-e a fejezetben leírt implicit konverziókkal.

Kulcsmegfigyelés: a kifejezés operandusainak konverziója az adott művelet végrehajtása előtt történik, az értékadáskor pedig a már kiszámított eredmény igazodik a célváltozó típusához.

3.5. AlgoRythmics szál: változócsere



E fejezet AlgoRythmics-betétje azt mutatja be, miként tudjuk két változó értékét egymással kicserélni. A rendezési algoritmusok egyik alapművelete, hogy a rendezendő sorozat két eleme helyet cserél (lásd később). A tánckoreográfiák kidolgozásánál külön figyelmet igényelt megfelelő tánclépések társítása ehhez az alapművelethez.

Két változó által tárolt érték egymás közötti kicserélésének leprogramozása összetettebb feladat, mint két táncos helycseréje. Ezt a 3.1. ábra is szemlélteti.

A változók értékcseréjének standard megoldása egy további, harmadik változó bevonását feltételezi, és három értékadással valósítható meg. Ezt a módszert gyakran háromveder-módszernek is nevezik, ezért a harmadik változót a továbbiakban vedernek fogjuk hívni. Ahogy az ábrán a nyilak sorszáma is mutatja, az értékadások sorrendje kulcsfontosságú.



3.1. ábra. *(a) A Buborék-rendezés csángó néptáncában az a[3] és a[4] cellák 8-as és 7-es értékeket képviselő táncosai helyet cserélnek; (b) A* `v1` *és* `v2` *változók értékcseréje háromveder-módszerrel.*

```
int v1 = 3, v2 = 5;
int veder;
veder = v1;  // v1 értékét kimásoljuk a veder-be
v1 = v2;     // v2 értékét átmásoljuk v1-be
v2 = veder;  // v2-be bemásoljuk a veder-ben tárolt értéket
printf("%i %i\n", v1, v2); // a képernyőn: 5 3
```

Ez a példa azt is hangsúlyozza, hogy az értékadást nem úgy kell elképzelni, mintha egy értéket „átmozgatnánk” egyik dobozból a másikba, hanem inkább úgy, hogy a bal oldalon szereplő változó értéke felülíródik a jobb oldali változó/kifejezés értékével, miközben a jobb oldalon szereplő változó értéke változatlan marad.

Az előzőekben bemutatott háromveder-módszer mellett létezik olyan megoldás is, amelyben a két változó értékének felcserélése további változó bevonása nélkül valósul meg. Az alábbi kódrészlet ezt a megoldást mutatja be.

```
int v1 = 3, v2 = 5;
v1 = v1 + v2;
v2 = v1 - v2;
v1 = v1 - v2;
printf("%i %i\n", v1, v2);   // a képernyőn: 5 3
```

A működés részletes magyarázatát most az olvasóra bízzuk. Érdemes a kódsorokat egyenként végigkövetni, és minden lépés után feljegyezni a v1 és v2 aktuális értékét. A megoldás megértése jól szemlélteti, mennyire fontos az értékadás pontos jelentésének ismerete.

[i] Megjegyzés. A programozásban gyakran találkozunk olyan megoldásokkal, amelyek első pillantásra „varázslatnak” tűnnek. Ezek azonban mindig következetes szabályokra épülnek – a kihívás az, hogy ezeket felismerjük.



AI-csepp: indoklás
Kérdezd meg az AI-t: melyik megoldás tekinthető „jobbnak”: a háromveder-módszer vagy a segédváltozó nélküli csere? Miért?
• Írj 3-4 mondatos saját álláspontot, és indokold meg a választásodat.
Tanulság: a változócsere nem „csere”, hanem másolások sorozata.

3.6. BBC szál: csoki vagy csili



A *The Secret Rules of Modern Living: Algorithms* című dokumentumfilm elején Marcus du Sautoy egy látszólag játékos, hétköznapi helyzeten keresztül vezeti be az algoritmikus gondolkodás fogalmát. Nem számítógépekkel, nem kóddal, hanem egy egyszerű társasjátékkal mutatja meg, hogy az algoritmus lényege nem a technológiában, hanem a stratégiában és a gondolkodásmódban rejlik.

A jelenetben egy üvegtálban csokoládék és egy csilipaprika található (3.2. ábra). A szabályok szándékosan egyszerűek: a játékosok felváltva húznak, minden lépésben meghatározott számú csokoládét vehetnek el, és az veszít, akire végül a csilipaprika marad. A helyzet ismerős, szinte gyermeki – mégis hamar kiderül, hogy a játék kimenetele nem a szerencsén múlik.

A filmjelenetben az üvegtálban 13 csokoládé található, amelynek számát Marcus, a kihívó ismeri, ellenfele, a diáklány azonban nem (3.2. ábra). Minden lépésben 1, 2 vagy 3 csokoládét vehet el az aktuális játékos. A játékot a matematikaprofesszor kezdi.



(a)



(b)

3.2. ábra. *(a) A játékosok felváltva húznak az üvegtálból a csoki vagy csili játék során; (b) A játék kezdeti állapota: 13 csokoládé és 1 csilipaprika.*

A kulcsmegfigyelés az, hogy a játékban kritikus állapotok léteznek. Ha a tálban lévő csokoládék száma a 4 többszöröse (például 12, 8 vagy 4), akkor az aktuális játékos vesztésre áll. Bármennyit is vesz el (1, 2 vagy 3 csokoládét), az ellenfél mindig képes úgy válaszolni, hogy a következő kör végére ismét a 4 többszöröse maradjon.

Mivel kezdetben 13 csokoládé van a tálban, az első játékos nyerő helyzetben van. Ha első lépésként 1 csokoládét vesz el, akkor 12 marad, ami már kritikus állapot. Ettől kezdve a stratégia egyszerű: minden lépésben annyit vesz el, hogy a két egymást követő húzás összesen 4 csokoládét távolítson el. Így a csokoládék száma sorban: 12 → 8 → 4 → 0 lesz, és végül a csilipaprika az ellenfélre marad.

A BBC filmjének ez a jelenete a következő programozási feladatot inspirálta: írj C programot, amely a fenti játék 13 csokoládés változatát szimulálja. A programban a felhasználó a számítógép ellen játszik. A játék kezdetén 13 csokoládé van a tálban, a csilipaprika pedig külön szerepel. A gép kezdi a játékot, és minden lépésben 1, 2 vagy 3 csokoládét vesz el. Ezután a felhasználó következik, aki szintén 1-3 csokoládét vehet el.

A gép stratégiája az algoritmikus gondolkodásból ismert szabályt követi: minden kör végén arra törekszik, hogy a megmaradó csokoládék száma a 4 többszöröse legyen.

Az alábbi C kód felfedezését az olvasóra bízzuk; a megoldás átlátása remélhetőleg tovább mélyíti az e fejezetben tanult ismereteket.

```
#include <stdio.h>

int main(){
    int csoki = 13;
    int user;

    printf("Csilis csoki jatek\n");
    printf("Kezdo csokoladek szama: %i\n\n", csoki);

    /* Gép 1. lépése: 1-et vesz el -> 12 marad */
    printf("A gep elvesz 1 csokoladet.\n");
    csoki = csoki - 1;
    printf("Maradt: %i\n\n", csoki);

    /* Felhasználó 1. lépése */
    printf("Te hany csokoladet veszel el (1-3)? ");
    scanf("%i", &user);
    csoki = csoki - user;
    printf("Maradt: %i\n\n", csoki);

    /* Gép 2. lépése: kiegészít 4-re */
```

```
    printf("A gep elvesz %i csokoladet.\n", 4 - user);
    csoki = csoki - (4 - user);
    printf("Maradt: %i\n\n", csoki);

    /* Felhasználó 2. lépése */
    printf("Te hany csokoladet veszel el (1-3)? ");
    scanf("%i", &user);
    csoki = csoki - user;
    printf("Maradt: %i\n\n", csoki);

    /* Gép 3. lépése */
    printf("A gep elvesz %i csokoladet.\n", 4 - user);
    csoki = csoki - (4 - user);
    printf("Maradt: %i\n\n", csoki);

    /* Felhasználó 3. lépése */
    printf("Te hany csokoladet vesz el (1-3)? ");
    scanf("%i", &user);
    csoki = csoki - user;
    printf("Maradt: %i\n\n", csoki);

    /* Gép utolsó lépése */
    printf("A gep elvesz %i csokoladet.\n", csoki);
    csoki = 0;
    printf("Maradt: %i\n\n", csoki);

    printf("A csilipaprika Rad marad.\n");
    printf("A gep nyert.\n");

    return 0;
}
```

Egy következő fejezetben a feladatot általánosabban is megoldjuk, úgy, hogy tetszőleges számú cukorkára működjön.

Az általánosítás egy része már a jelenlegi tudásunkkal is megoldható. Írj programot, amely tetszőleges, a bemenetről beolvasott `n` cukorkaszám esetén kiírja:

- hány forduló szükséges a játék végéig, ha a tanár optimális kezdőlépést tesz, majd a „kiegészítés 4-re” stratégiát követi;
- mekkora legyen a tanár kezdő lépése, vagyis hány cukorkát vegyen el indításként.

Ráadás kérdés: mely `n` értékekre nem működne a megbeszélt stratégia?



AI-csepp: stratégiaalkotás

Kérdezd meg az AI-t: milyen kérdéseket kell tisztázni, mielőtt stratégiát alkotunk egy ilyen játékra?

- Ne megoldást kérj, hanem kérdések listáját!

Algoritmikai tanulság:

- a stratégia nem a lépésekből, hanem az állapotok felismeréséből születik;
- a programozás nem csak szintaxis, ugyanolyan fontos az algoritmikus gondolkodás is.

3.7. Hiúz szál: barlangátkelés



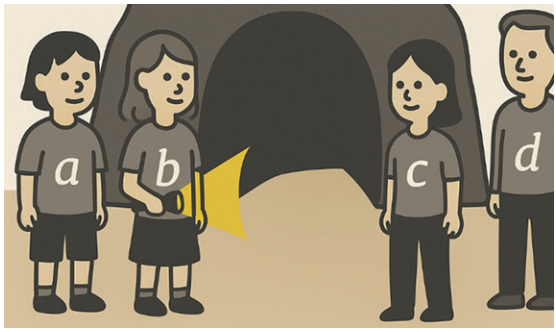
Egy négytagú család szeretne átkelni egy sötét barlangon (3.3. ábra). Mindössze egy zseblámpájuk van, és egyszerre legfeljebb ketten mehetnek. Ha ketten mennek, az átkelés ideje a lassabbik ideje. Mennyi idő alatt juthat át a család a lehető leggyorsabban? A négy családtag átkelési ideje: $a \leq b \leq c \leq d$ (percben). (ELTE 2026)

Milyen stratégiák jöhetnek szóba?

1. a leggyorsabb (a) futárkodik (összidő: $T1$);
2. a két legidősebb (c és d) együtt kel át (összidő: $T2$).

A megfelelő stratégiák például az alábbiak (a nyilak azt jelzik, hogy oda- vagy visszaútról van szó; aláhúztuk, hogy az egyes átkelésekkor melyik idő számít):

1. $[a + \underline{d}]\ (\rightarrow), [\underline{a}]\ (\leftarrow), [a + \underline{c}]\ (\rightarrow), [\underline{a}]\ (\leftarrow), [a + \underline{b}]\ (\rightarrow)$;
2. $[a + \underline{b}]\ (\rightarrow), [\underline{a}]\ (\leftarrow), [c + \underline{d}]\ (\rightarrow), [\underline{b}]\ (\leftarrow), [a + \underline{b}]\ (\rightarrow)$;



3.3. ábra. *Négytagú család egy elemlámpás átkelése egy szűk barlangon*

Írj programot, amely kiszámolja $T1$ és $T2$ értékét! A program:

- olvassa be `a`, `b`, `c`, `d` értékeit billentyűzetről,

• számítsa ki képlettel $T1$ és $T2$ értékét,
• és írja ki mindkettőt a képernyőre.

Adj meg *három különböző bemenetet*, amelyek mindhárom helyzetet megmutatják:

• (i) amikor $T1$ a jobb,
• (ii) amikor $T2$ a jobb,
• (iii) amikor döntetlen.

Ez a feladat is általánosítható, például tetszőleges számú családtagra kiterjesztve.



AI-csepp: rejtett feltételezések és határesetek

Kérdezd meg az AI-t: milyen rejtett feltételezéseket tartalmaz a barlangátkelés feladat megfogalmazása és a képletekkel számolt megoldás?

• A felsorolás alapján válaszold meg: (i) mely feltételezések szükségesek, (ii) és melyek azok, amelyeket a programnak ellenőriznie kellene.

Tanulság: a legtöbb hiba nem ott van, amit leírtunk, hanem ott, amit nem.

3.8. ECN szál: Piroska és a farkas



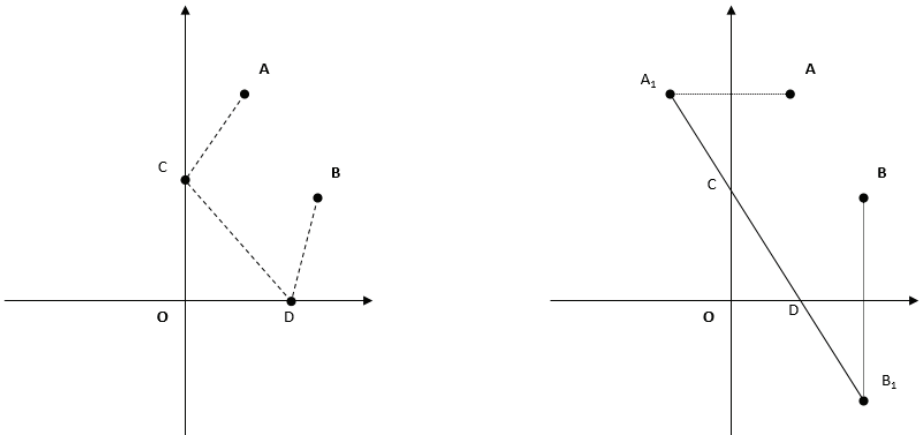
Írj programot, amely a billentyűzetről beolvassa két pont (A és B) koordinátáit (az euklideszi sík első negyedébe esnek; A első szögfelező felett, B első szögfelező alatt), és kiírja a képernyőre az A-ból B-be vezető azon legrövidebb út hosszát, amely érinti az Ox és Oy egyeneseket (3.4. ábra).

Ezt a feladatot a DEIK (Debreceni Egyetem Informatikai Kara) regionális programozó csapatversenyen tűzték ki, eredeti, „mesés” megfogalmazásban (DEIK 2026). Eszerint Piroska az A pontban található, a nagymamája pedig a B pontban. Piroska a legrövidebb úton szeretne eljutni a nagymamához úgy, hogy közben érinti mindkét tengelyt (az Ox-et és az Oy-t), amelyek a történetben patakokat jelképeznek. A patakoknál vizet vesz, hogy vihessen a nagymamájának.

A kihívás fokozása érdekében nem közöljük közvetlenül a megoldást, hanem egy ábra segítségével adunk hozzá ötletet. Emellett megadjuk a megoldás C nyelvű kódját is, amelyből az olvasó kódolvasással és annak megfejtésével meríthet további tanulságokat.

A képaláírás sugallta algoritmus:

1. tükrözzük A pontot az Oy tengelyre (legyen A_1)
2. tükrözzük B pontot az Ox tengelyre (legyen B_1)
3. számítsuk ki az A_1 és B_1 közti távolságot (Pitagorasz tétele)



(a) (b)

3.4. ábra. *(a) A feladatot szemléltető ábra: a C és D pontok azon pozícióját keressük, amelyre az |AC|+|CD|+|DB| távolság minimális. (b) A megoldást szemléltető ábra: ha A_1 az A pont szimmetrikusa az Oy tengelyre, B_1 pedig a B pont szimmetrikusa az Ox tengelyre, akkor $|A_1C|+|CD|+|DB_1|=|AC|+|CD|+|DB|$. Az $|A_1C|+|CD|+|DB_1|$ összeg nyilván akkor minimális, ha az A_1, C, D és B_1 pontok egy egyenesre esnek. Ebben az esetben a keresett távolság Pitagorasz tétele segítségével határozható meg.*

A *Piroska és a farkas* feladat megoldásának egy lehetséges C kódja:

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

int main() {
    float xA, yA, xB, yB;
    scanf("%f %f %f %f", &xA, &yA, &xB, &yB);

    float dx = xA + xB;
    float dy = yA + yB;
    float d = sqrt(dx * dx + dy * dy);

    printf("%f\n", d);
    return 0;
}
```

A fenti megoldásból az is kiderül, hogy a gyökvonásra létezik könyvtári függvény (`sqrt`), amelynek használata feltételezi az `#include` <math.h> programsort.



AI-csepp: nézőpontváltás

Kérdezd meg az AI-t: hogyan lehet erre a megoldásra rájönni, ha még nem ismerjük a trükköt?

- A válasz alapján próbáld meg felsorolni a gondolkodási lépéseket.
- Írd le, hogy miért találod „szépnek” ezt a megoldást!

Tanulság: a megoldás kulcsa gyakran nem a kód, hanem a nézőpontváltás.

▶▶ Zárógondolat

A számítógép számára minden program végső soron bitek sorozata, a programozó azonban nem bitekkel dolgozik, hanem változókkal, értékekkel és utasításokkal.

Ebben a fejezetben megtettük az első lépéseket ezen az úton: megismertük, miként tárolunk adatokat, hogyan végzünk rajtuk műveleteket, és hogyan lép kapcsolatba a program a külvilággal. Bár az eddigi példák egyszerűnek tűnhetnek, valójában már most minden olyan eszköz a rendelkezésünkre áll, amelyre egy működő program felépítéséhez szükség van: adatokat tudunk tárolni, módosítani, beolvasni és megjeleníteni.

A programozás lényege nem pusztán új utasítások megtanulása, hanem ezeknek az elemeknek az átgondolt kombinálása. A következő fejezetben azt vizsgáljuk meg, miként köthetjük a program végrehajtását feltételekhez és ismétlődésekhez a C nyelv utasításkészlete segítségével.

4. Utasításkészlet: kevés darab, végtelen kombináció

A C nyelv utasításkészlete meglepően kicsi. Valójában néhány alapvető utasítás elegendő ahhoz, hogy egyszerű és rendkívül összetett programokat egyaránt felépítsünk. A programozás kihívása nem az utasítások számában, hanem az ezekből való építkezésben rejlik.

Ennek megértéséhez érdemes nem magából a nyelvből, hanem a programozás általános gondolkodási modelljéből kiindulni.

4.1. A vezérlés három alapformája

A strukturált programozás egyik alapvető felismerése, hogy minden algoritmus felépíthető három vezérlési szerkezet segítségével:

- szekvencia,
- elágazás,
- ismétlés.

Ezek nem egy konkrét programozási nyelv sajátosságai, hanem általános gondolkodási minták, amelyek bármely algoritmusban megjelennek. Egy program végrehajtása ezek kombinációjaként írható le.

- A szekvencia egymás után végrehajtott lépések sorozata.
- Az elágazás lehetővé teszi, hogy egy feltételtől függően különböző végrehajtási utak közül válasszunk.
- Az ismétlés pedig ugyanazon lépések többszöri végrehajtását teszi lehetővé.

A C nyelv utasításkészlete pontosan ezeknek a vezérlési formáknak a leírására szolgál.

4.2. A C nyelv alapvető utasításai

A C nyelvben nyolc alapvető utasítást különböztetünk meg, amelyek négy csoportba sorolhatók.

- Kifejezés-utasítás
- Döntési utasítások: `if`, `switch`
- Ismétlési utasítások: `for`, `while`, `do-while`
- Ugró utasítások: `break`, `continue`

Ez az utasításkészlet első látásra szűkösnek tűnhet, de pontosan elegendő ahhoz, hogy a három alapvető vezérlési szerkezet megvalósítható legyen.

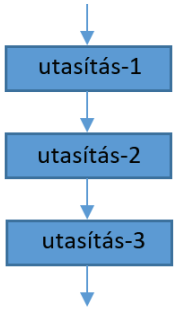
4.2.1. Kifejezés-utasítás

A kifejezés fogalmával már eddig is találkoztunk. A kifejezés-utasítás egy pontosvesszővel lezárt kifejezés. Ennek leggyakoribb formája az értékadás, amelyet az előző fejezetben mutattunk be. Bár ott azt mondtuk, hogy az értékadás jobb oldalán szerepelhet egy kifejezés, mivel az = operátor C-ben maga is operátornak számít, ezért az értékadás mint egész is kifejezésnek tekinthető.

A függvényhívások – amelyek részfeladatok lebonyolításáért felelősek – szintén kifejezés-utasításnak számítanak. Ilyenek például a `scanf`, `printf` vagy `freopen` függvényhívásokat tartalmazó programsorok.

Mindezekre tekintettel kijelenthető, hogy ez az utasítástípus a „munka” elvégzéséért felel: számolásért, változtatásért és műveletek végrehajtásáért.

Eddigi programjaink túlnyomórészt kifejezés-utasításokat tartalmaztak, ezek sorozataként kerültek lekódolásra. Ezek a kódok szerkezetileg még csak szekvenciát tartalmaztak (4.1. ábra).



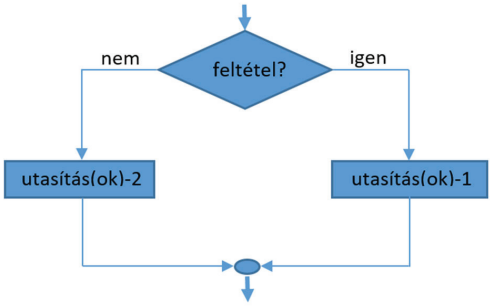
4.1. ábra. *Szekvencia: egymás után végrehajtott lépések sorozata*

A 10. fejezetnek mint mesterfejezetnek az egyik kulcsfogalma a kifejezés lesz, ahol további részletek tárulnak majd fel a kifejezés-utasítás működéséről és szerepéről is.

4.2.2. Döntési utasítások – párhuzamos forgatókönyvek

Egy program tekinthető forgatókönyvnek egy feladat megoldásához. A döntési vagy elágazási utasítások lehetővé teszik, hogy párhuzamos forgatókönyveket adjunk meg: attól függően, hogy például egy feltétel igaz vagy hamis, ez vagy az az utasítás, illetve utasítássor kerül végrehajtásra.

A C nyelvben ezt a döntést az `if` és a `switch` utasítások segítségével valósíthatjuk meg; ezek feleltethetők meg az elágazás vezérlési szerkezetének. Mindkét esetben egy kifejezés kiértékelése határozza meg, melyik végrehajtási ág aktiválódik. A döntési utasítások tehát nem önálló „munkát” végeznek, hanem irányítják az utasítások végrehajtását (4.2. ábra).




```
if ( <feltétel> ) {
   <utasítás(ok)-1>
} else {
   <utasítás(ok)-2>
}
```

4.2. ábra. *Elágazási vezérlési szerkezet: logikai sémával és C utasítással. Attól függően, hogy a feltétel igaz vagy hamis,* `<utasítás(ok)-1>` *vagy* `<utasítás(ok)-2>` *kerül végrehajtásra.*

Megjegyzendő, hogy bár eddigi programjaink még nem tartalmaztak explicit döntési utasításokat, számos helyzetben már lett volna helyük. Gondoljunk például a `scanf` függvény használatára. A `scanf` visszatérési értéke jelzi, hogy az adatbeolvasás sikeres volt-e vagy sem. Ez valójában két lehetséges forgatókönyvet jelent:

- a beolvasás sikerült, a program folytathatja a számítást;
- a beolvasás sikertelen, a programnak másképp kellene reagálnia.

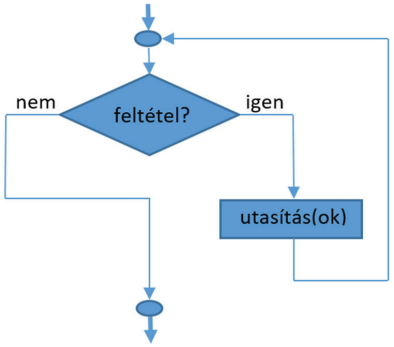
Döntési utasítások hiányában eddig feltételeztük, hogy a kedvező forgatókönyv valósul meg. A döntési utasítások bevezetésével azonban a programozó már nemcsak feltételez, hanem tudatosan kezeli a lehetséges helyzeteket, és a program a futás közben felmerülő körülményekhez igazodik. Az eredmény egy robusztusabb program, amely bármely lehetséges szituációban „feltalálja magát”.

4.2.3. Ismétlési utasítások – ugyanaz újra és újra

A programozás során gyakran találkozunk olyan helyzetekkel, amikor ugyanazt a feladatot nem egyszer, hanem többször kell elvégezni. Ilyenkor nem új forgatókönyvre van szükség, hanem arra, hogy egy már meglévő forgatókönyv ismételten lefusson.

Az ismétlési, más szóval ciklusutasítások ezt a célt szolgálják: az ismétlés vezérlési szerkezetét valósítják meg. A C nyelvben a `for`, `while` és `do-while` utasítások teszik lehetővé, hogy egy vagy több utasítás addig ismétlődjön, amíg egy feltétel teljesül, vagy egy előre meghatározott számú lépés le nem zajlik. Itt

is a vezérlésről van szó: nem arról, hogy mit csinál a program, hanem arról, hogy hányszor és meddig (4.3. ábra).




```
while ( <feltétel> ) {
    <utasítás(ok)>
}
```

4.3. ábra. *Ismétlési vezérlési szerkezet: logikai sémával és C utasítással. Amíg a feltétel igaz, újra és újra végrehajtásra kerül* `<utasítás(ok)>`.

Korábbi feladatainkban már találkoztunk olyan helyzettel, ahol az ismétlés igénye megjelent. A *csoki vagy csili* feladat megoldásában például ugyanazt a kódrészletet többször is leírtuk, ahelyett, hogy azt mondtuk volna a programnak, mit kell ismételnie és hányszor.

Ebben az esetben az ismétlés kézzel történt. Ez működő megoldás, de nehezen általánosítható. Az ismétlési utasítások bevezetésével a programozó már szabályt fogalmaz meg, nem pedig másolatokat készít.

4.2.4. Ugró utasítások – beavatkozás a végrehajtásba

Az eddig bemutatott vezérlési szerkezetek – a döntések és az ismétlések – jellemzően egy előre meghatározott forgatókönyvet követnek. Előfordulhat azonban, hogy a program végrehajtása közben rendkívüli helyzet áll elő, amelyben a szokásos menet megszakítása vagy módosítása indokolt.

Az ugró utasítások ilyen esetekben adnak lehetőséget a programozónak arra, hogy közvetlenül beavatkozzon a végrehajtás menetébe. A C nyelvben idetartozik a `break` és a `continue` utasítás.

Fontos hangsúlyozni, hogy ezek nem önálló vezérlési szerkezetek. Nem helyettesítik az elágazást vagy az ismétlést, hanem azokkal együtt használjuk őket, működésüket módosítják. Használatuk ezért körültekintést igényel: túlzott alkalmazásuk könnyen átláthatatlanná teheti a program vezérlését. Megfelelő helyen és mértékkel alkalmazva azonban az ugró utasítások egyszerűbbé és érthetőbbé tehetik a programot, mivel világosan kifejezik a kivételes helyzeteket és a megszakítás okát.



AI-csepp: kevés darab, végtelen kombináció

Kérdezd meg az AI-t: hogyan lehet ugyanazokból az alapvető vezérlési szerkezetekből és utasításokból egyszerű és rendkívül összetett programokat is felépíteni?

A válaszban figyeld meg:

- mire helyezi a hangsúlyt (utasítások száma vagy szervezésük),
- hogyan jelenik meg az egyszerű és az összetett rendszer közötti különbség,
- és milyen analógiákat használ a magyarázathoz.

Ezután válaszolj saját szavaiddal: miben különbözik egy „kutyaól” jellegű program egy „metropolisz” jellegűtől, ha ugyanazok az építőelemek állnak rendelkezésre?

⯅Betét: valódi játékprogram belülről

Az alábbi kódrészlet egy nyílt forráskódú, közkedvelt játékprogram egyik belső részlete. Bár a teljes program több tízezer sorból áll, ez a néhány sor – még ha tartalmaz is számunkra egyelőre ismeretlen elemeket – jól mutatja, hogy az összetett működés ugyanazokra az alapvető vezérlési szerkezetekre és utasításokra épül, amelyeket ebben a fejezetben megismertünk.

```
ball.x += ball.vx;
ball.y += ball.vy;
if (ball.x < 0 || ball.x > max_x) {
    ball.vx = -ball.vx;
}
for (int i = 0; i < bubble_count; i++) {
    bubbles[i].y += bubbles[i].speed;
}
```

A kódrészlet nem grafikai megjelenítést, hanem játékmeneti logikát ír le: mozgást, döntést és ismétlést. Megfigyelhető, hogy a vezérlés itt is kizárólag kifejezés-utasításokra, döntésekre és ciklusokra épül.

Egy ilyen program nem azért összetett, mert „többféle utasítást” használ, hanem azért, mert sok egyszerű részlet szerveződik egymásra.

►► Zárógondolat

A C nyelv utasításkészlete szándékosan letisztult. Nem az utasítások sokfélesége adja az erejét, hanem az a tény, hogy néhány alapvető elem megfelelő kombinációjával tetszőlegesen összetett programok építhetők fel.

Ez a fejezet felülnézeti képet adott a C nyelv utasításkészletéről, és kóstolóul is szolgált egy-két utasítás tekintetében. A következő fejezetekben először a döntési utasításokat vizsgáljuk meg részletesen, majd az ismétlések segítségével tanuljuk meg, miként válhat a program valóban alkalmazkodóvá és általánosíthatóvá.

5. Elágazás a programban – döntési utasítás

Az eddigi programok végrehajtása egyetlen, előre rögzített útvonalat követett. Az utasítások sorban hajtódtak végre, feltételezve, hogy minden a „megfelelő módon" történik. A valós feladatok megoldása során azonban gyakran több lehetséges folytatás is elképzelhető, és a programnak futás közben kell döntenie arról, melyik irányba haladjon tovább.

A döntési, más szóval elágazási utasítások teszik lehetővé, hogy a program feltételek alapján válasszon a különböző végrehajtási lehetőségek közül. Egy ilyen döntési ponton párhuzamos forgatókönyvek közül pontosan egy valósul meg, attól függően, hogy egy kifejezés kiértékelése igaz vagy hamis eredményt ad. Ily módon a program vezérlése már nem csupán utasítások sorozata, hanem alkalmazkodó döntési folyamat lesz.



Ebben a fejezetben az `if` utasítást vizsgáljuk meg, amely általános feltételek kezelésére alkalmas, és a feladatok túlnyomó többségében ezzel valósítjuk meg az elágazás vezérlési szerkezetét. A `switch` utasítást – amely egy egész kifejezés lehetséges értékei közötti választást támogatja – egy későbbi fejezetben mutatjuk be.

5.1. Az `if` utasítás – kétutas elágazás

Az `if` utasítás feltételes végrehajtást valósít meg: a program egy feltételt (logikai kifejezést) értékel ki, és ennek eredménye alapján választ a lehetséges folytatások közül. Általában nem elég csak azt megmondani, mi történjen akkor, ha a feltétel teljesül (igaz), hanem azt is szeretnénk rögzíteni, mi történjen akkor, ha nem (hamis). Ezt nevezzük kétutas elágazásnak.

</> Szintaxis: a kétutas elágazás általános alakja

```
if ( <feltétel> ) {
    /* igaz ag */
} else {
    /* hamis ag */
}
```

Első példaként tekintsük azt a feladatot, amikor a billentyűzetről beolvasunk két egész számot, és ezeket növekvő sorrendben szeretnénk kiíratni a képernyőre.

Ha a két számot az `x` és `y` változókba olvassuk be, akkor a megoldás valójában két lehetséges kiíratás egyikének kiválasztását jelenti: vagy az `x y`, vagy az `y x` sorrendet kell megjeleníteni. A két forgatókönyv közül az hajtódik végre, amely megfelel annak, hogy az `x` vagy az `y` változó tartalmazza-e a nagyobb értéket. Ennek eldöntésére használjuk az `if` utasítást.

Mindezek figyelembevételével a feladat megoldása az alábbi kódrészlet.

```
int x, y;

scanf("%i %i", &x, &y);

if ( x <= y ) {
    printf("%i %i\n", x, y);
} else {
    printf("%i %i\n", y, x);
}
```

Már egy olyan egyszerű feladat is, mint egy szám beolvasása, feltételezheti párhuzamos forgatókönyvek megadását, ha igényesebben szeretnénk leprogramozni. Eddigi példáinkban feltételeztük, hogy a `scanf` általi beolvasás sikeres lesz – de mi történik akkor, ha ez nem így van?

A `scanf` függvény a beolvasás lebonyolításán túl visszatéríti a sikeresen beolvasott értékek számát. Ennek felhasználásával robusztusabb megoldások készíthetők, amelyek a hibás beolvasást is kezelik. Amennyiben nem adunk meg külön forgatókönyvet arra az esetre, ha a beolvasás sikertelen, a program működése bizonytalanná válhat.

```
int x;
if ( scanf("%i", &x) == 1 ) {
    printf("Sikeres beolvasas: x = %i\n", x);
    // a beolvasott ertek feldolgozasa
} else {
    printf("Hibas beolvasas.\n");
}
```

Itt is két párhuzamos forgatókönyvet adtunk meg: (i) sikeres beolvasás esetén a program folytathatja az adatfeldolgozást; (ii) hiba esetén másképp reagál (például jelez és leáll, vagy később megtanuljuk, hogy miként kérheti újra a bemeneti adatokat).

5.2. Egyutas elágazás

Nem mindig van szükség két ágra. Sokszor csak egy kivételes vagy hibás helyzetet szeretnénk kezelni, és ha a feltétel nem teljesül, a program egyszerűen megy tovább. Ezt nevezzük egyutas elágazásnak.

</> Szintaxis: az egyutas elágazás általános alakja

```
if ( <feltétel> ) {
    /* csak igaz esetben vegrehajtando */
}
```

A két szám növekvő sorrendbe való kiírásának feladata egyutas elágazásos szerkezettel is megoldható. Az alábbi kód ezt mutatja be. Az elemzését az olvasóra hagyjuk.

```
#include <stdio.h>

int main(){
    int x, y, veder;

    scanf("%i %i", &x, &y);

    /* Ha nem megfelelő a sorrend, felcseréljük az értékeket */
    if (x > y) {
        veder = x;
        x = y;
        y = veder;
    }

    /* A kiírás már mindig növekvő sorrendben történik */
    printf("%i %i\n", x, y);

    return 0;
}
```

Egy további példa egyutas `if` utasítás használatára a `freopen` ellenőrzése. A `freopen` függvény tipikusan fájlok átirányítására szolgál. Ha a művelet nem sikerül, a függvény `NULL` értékkel tér vissza. Ilyenkor érdemes rögtön kezelni a hibát.

```
if ( freopen("input.txt", "r", stdin) == NULL ) {
    printf("Nem sikerult megnyitni az input.txt fajlt.\n");
    return 0;
}
```

Ha a fájl megnyitása sikerül, nincs külön teendő: a program folytatódik, és a további `scanf` hívások már a fájlból olvasnak. Ellenkező esetben értelmetlen a folytatás, ezért egy megfelelő üzenetet követően a program leáll (`return 0;`).

5.3. Összehasonlítási operátorok

Az `if` zárójelében álló feltétel sokszor két érték összehasonlításán alapul. A C nyelvben az összehasonlítás eredménye logikai jellegű: a kifejezés vagy „igaz”, vagy „hamis”. Bár matematikából ismerősek lehetnek az összehasonlítási operátorok, már felfigyelhettünk arra, hogy programozási környezetben egyeseket másképpen írunk. Az 5.1. táblázat a C nyelv összehasonlítási operátorait mutatja be.

5.1. táblázat. *A C nyelv összehasonlítási operátorai*

Operátor	Jelentés	Példa kifejezés	Magyarázat
==	egyenlő	`x == y`	igaz, ha `x` és `y` értéke megegyezik
!=	nem egyenlő	`x != y`	igaz, ha `x` és `y` különbözik
<	kisebb	`x < y`	igaz, ha `x` kisebb, mint `y`
<=	kisebb vagy egyenlő	`x <= y`	igaz, ha `x` kisebb vagy egyenlő `y`-nal
>	nagyobb	`x > y`	igaz, ha `x` nagyobb, mint `y`
>=	nagyobb vagy egyenlő	`x >= y`	igaz, ha `x` nagyobb vagy egyenlő `y`-nal

Különösen fontos odafigyelni arra, hogy az egyenlőség vizsgálata `==` által történik, ugyanis az `=` az értékadásra szolgál.

5.4. Logikai operátorok

Ahogyan a numerikus értékekkel aritmetikai műveletek végezhetők, úgy a logikai értékekkel – igaz vagy hamis – logikai műveletek hajthatók végre. Ezek segítségével egyszerű feltételekből összetettebb feltételek építhetők fel.

A C nyelv három alapvető logikai operátort biztosít: a logikai ÉS-t, a logikai VAGY-t, valamint a logikai NEM-et (tagadást). Ezek rendre `&&`, `||` és `!` jelekkel írhatók le.

A logikai operátorok működését az 5.2. táblázat foglalja össze, amelyek megmutatják, hogy az egyes műveletek milyen eredményt adnak az operandusok igaz vagy hamis értékei esetén. Vegyük észre, hogy az ÉS eredménye akkor igaz, ha mindkét operandus értéke igaz, a VAGY akkor, ha valamelyik operandus igaz.

5.2. táblázat. *A logikai operátorok működése*

Logikai ÉS (`&&`)

A	B	A && B
hamis	hamis	hamis
hamis	igaz	hamis
igaz	hamis	hamis
igaz	igaz	igaz

Logikai VAGY (`||`)

A	B	A || B
hamis	hamis	hamis
hamis	igaz	igaz
igaz	hamis	igaz
igaz	igaz	igaz

Logikai NEM (tagadás) (`!`)

A	!A
hamis	igaz
igaz	hamis

A logikai operátorokkal több feltétel összekapcsolható. Például ha azt szeretnénk ellenőrizni, hogy egy szám két határ között van, gyakran `&&` operátort használunk; ha pedig azt, hogy két lehetőség közül valamelyik teljesül, `||` operátort.

- Intervallumvizsgálat: `( x >= a && x <= b )`
- Két különböző elfogadott érték: `( x == 0 || x == 1 )`

A szökőév ellenőrzése jó példa a logikai operátorok használatának hasznára, valamint arra, hogyan írható le egy valós szabályrendszer pontosan egyetlen összetett logikai kifejezés segítségével. Egy év szökőév, ha osztható 4-gyel, ÉS nem osztható 100-zal, VAGY osztható 400-zal.

```
#include <stdio.h>

int main(){
    int ev;

    scanf("%i", &ev);

    if ( (ev % 4 == 0 && ev % 100 != 0) || (ev % 400 == 0) ) {
```

```
            printf("%i szokoev.\n", ev);
        } else {
            printf("%i nem szokoev.\n", ev);
        }

        return 0;
    }
```

5.5. Egymásba ágyazott `if` – több kizáró eset kezelése

Vannak feladatok, ahol nem két, hanem több egymást kizáró eset közül kell választani. Ilyenkor az `if` utasítások egymásba ágyazhatók, illetve egymás után fűzhetők. Klasszikus példa erre az egyenletek megoldása, ahol a lehetséges esetek száma többféle.

Tekintsük a másodfokú egyenletet, amelynek jól ismert általános alakja: $ax^2+bx+c=0$.

Egy másodfokú egyenletet megoldó program bemeneti adatai a három együttható (a, b, c) értékei. Itt azonban rögtön felmerül néhány fontos kérdés.

- Mi van akkor, ha $a = 0$? Ebben az esetben az egyenlet már nem másodfokú, hanem elsőfokú: $bx + c = 0$.
- És mi van akkor, ha $b = 0$ is? Akkor az egyenlet konstans: $c = 0$.

Ezért a robusztus megoldásnak ezeket az eseteket is kezelnie kell.

Az alábbi program ezt az egymásba ágyazott döntési logikát mutatja be. A vezérlés több szinten jelenik meg: először azt döntjük el, hogy valóban másodfokú-e az egyenlet, majd a megfelelő ágon belül további eseteket különböztetünk meg.

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

int main(){
    float a, b, c;
    scanf("%f %f %f", &a, &b, &c);

    if (a == 0) {
        /* Nem masodfoku, hanem elsofoku vagy konstans */
        if (b == 0) {
            /* Konstans egyenlet: c = 0 */
            if (c == 0) {
                printf("Vegtelen sok megoldas van.\n");
            } else {
                printf("Nincs megoldas.\n");
            }
        } else {
```

```
            /* Elsofoku: bx + c = 0 */
            float x = -c / b;
            printf("Elsofoku egyenlet: x = %f\n", x);
        }
    } else {
        /* Masodfoku: ax^2 + bx + c = 0 */
        float d = b*b - 4*a*c;  /* diszkriminans */

        if (d < 0) {
            printf("Nincs valos megoldas.\n");
        } else if (d == 0) {
            float x = -b / (2*a);
            printf("Egy valos megoldas: x = %f\n", x);
        } else {
            /* Ket kulonbozo valos megoldas */
            float x1 = (-b + sqrtf(d)) / (2*a);
            float x2 = (-b - sqrtf(d)) / (2*a);
            printf("Megoldas: x1 = %f, x2 = %f\n", x1, x2);
        }
    }

    return 0;
}
```

[i] Megjegyzések

- A gyökvonást az `sqrt` (illetve `float` változók esetén az `sqrtf`) függvény végzi, amelynek használata feltételezi az `#include <math.h>` programsort.
- A példában több egyenlőségvizsgálat is szerepel. Valós számok esetén később visszatérünk arra, hogy miért lehet célszerű „közel nullához” jellegű ellenőrzést alkalmazni.

Ez az egymásba ágyazott szerkezet jól mutatja, hogy az `if` utasítással nemcsak két kimenetelt, hanem több egymást kizáró forgatókönyvet is kezelhetünk.

Gyakorló feladatok

- Írd meg a szökőév ellenőrzést egymásba ágyazott `if`-kel (logikai operátorok használata nélkül)!
- Biztonságos osztás: olvass be két egész számot (`szamlalo`, `nevezo`); ha a nevező `0`, írj ki hibaüzenetet, különben írd ki az egész osztás és a valós osztás eredményét.
- Életkor ellenőrzése és besorolása: a felhasználó megadja az életkorát; `0–17` esetén kiskorú, `18–64` esetén felnőtt, `65–130` esetén nyugdíjas, míg `0` alatti vagy `130` feletti érték hibás adat.

- Vásárlás hitelkerettel: olvasd be az egyenleget, a hitelkeretet és a vásárlás árát; ha az ár ≤ egyenleg, a vásárlás elfogadva, ha ≤ egyenleg + hitelkeret, elfogadva hitellel, különben elutasítva.
- Koncertbelépés időkorláttal: a program olvassa be az életkort és az aktuális órát (`0–23`); `18` év alatt csak `22` óra előtt lehet belépni, `18` éves kortól időkorlátozás nélkül.
- Hőmérséklet és páratartalom alapján döntés: olvasd be a hőmérsékletet és a páratartalmat; `18 °C` alatt fűtés, `25 °C` felett szellőztetés, `18–25 °C` között csak akkor szellőztetés, ha a páratartalom `70%` felett van.
- Jegy és minősítés: olvasd be a jegyet (`1–5`); ha a jegy érvényes, írd ki a minősítést is: `1` → elégtelen, `2` → elégséges, `3` → közepes, `4` → jó, `5` → jeles; minden más érték hibás adat.
- Késés megítélése igazolással: a felhasználó megadja a késés perceit és azt, hogy van-e igazolása (`0` = nincs, `1` = van); `0` perc pontos érkezés, `1–5` perc kis késés, `6` perc vagy több jelentős késés, de igazolás esetén a késés elfogadott.



AI-csepp: tipikus hibák – gondolkodási csapdák
Kérdezd meg az AI-t: milyen tipikus hibák fordulnak elő az alábbi témakörökben, és mire érdemes különösen odafigyelni a megoldás során?

- az = és == összekeverése;
- a feltétel „fejben” jó, de kódban hibás megfogalmazása;
- zárójelek elhagyása összetett feltételeknél;
- valós számok összehasonlítása == operátorral;
- feleslegesen bonyolult elágazási szerkezetek;
- elfelejtett határesetek;
- az `if` utasítás szerepének félreértése.

[i] Megjegyzés. A legtöbb hiba nem szintaktikai, hanem gondolkodási eredetű: a programozó nem pontosan fogalmazza meg, mit szeretne ellenőrizni. Ha egy feltételt nem tudunk egyértelműen megfogalmazni szóban, akkor kódban sem lesz megbízható.

5.6. BBC szál: csoki vagy csili 2.



A 3.7. alfejezetben bemutatott csoki vagy csili feladat kapcsán előrevetítettük, hogy később tetszőleges cukorkaszámra is megoldjuk a problémát. Ehhez még nem tudunk eleget, de ahhoz már igen, hogy egy adott bemenetre eldöntsük: a tanár kezdjen-e, vagy érdemesebb átengednie a kezdést a diáklánynak.

Feladat: Írd meg azt az `if` utasítást (illetve feltételt), amely leírja a tanár szerepét magára öltő gép optimális kezdőlépését, ha a játék a továbbiakban a „kiegészítés 4-re” stratégia szerint zajlik!

5.7. Hiúz szál: minimumkeresés négy elemre



A 3.8. alfejezetben bemutatott barlangátkelés feladatban rögzítettük, hogy a négy családtag átkelési ideje növekvő sorrendben adott. De mi történik akkor, ha ezek az időértékek tetszőleges sorrendben kerülnek beolvasásra? Ebben az esetben a programnak kell meghatároznia például azt, hogy melyik a leggyorsabb családtag.

Legyen tehát a feladat az, hogy határozzuk meg a család minimális átkelési idejét, amennyiben a leggyorsabb futárkodik stratégiát alkalmazzuk. Ebben az esetben az odautak ideje rendre a többi családtag átkelési idejének összege, ami összesen ($a + b + c + d - t_{leggyorsabb}$) időt jelent, míg a visszautak összideje $2 \cdot t_{leggyorsabb}$.

Az alábbi megoldás a legkisebb érték meghatározására a következő módszert alkalmazza:

- kezdetben az első értéket (`a`) tekintjük `t_legkisebb`-nek;
- ezután sorra összehasonlítjuk az összes többi értéket az aktuális `t_legkisebb` értékkel, és valahányszor kisebbet találunk, azt tesszük meg új `t_legkisebb`-nek.

Ezt az algoritmust építettük be az alábbi kódba.

```
#include <stdio.h>

int main(void)
{
    int a, b, c, d;
    int t_legkisebb;

    /* Beolvassuk a negy csaladtag atkelési idejet */
    scanf("%i %i %i %i", &a, &b, &c, &d);

    /* Kezdetben az elso erteket tekintjuk a legkisebbnek */
    t_legkisebb = a;

    /* Sorra osszehasonlitjuk a tobbi ertekkel */
    if (b < t_legkisebb) {
        t_legkisebb = b;
    }

    if (c < t_legkisebb) {
```

```
            t_legkisebb = c;
        }

        if (d < t_legkisebb) {
            t_legkisebb = d;
        }

        /* Minimalis atkelési ido */
        int osszido = (a+b+c+d - t_legkisebb) + 2 * t_legkisebb;

        printf("Minimalis atkelési ido: %d\n", osszido);

        return 0;
    }
```

Figyeljük meg, hogy a minimum meghatározásához nem volt szükség rendezésre. A feladat megoldásához elegendő volt egymás után összehasonlítani az értékeket, és mindig a kisebbet „megtartani”. Ez az elv később ciklusok segítségével általánosítható tetszőleges elemszámra.

5.8. AlgoRythmics szál: három szám rendezése – buborékos, kiválasztó, beszúró



Ebben a fejezetben az `if` utasítással ismerkedünk, de már most érdemes megmutatni, hogy a döntésekből hogyan lesz „ritmus”: ismétlődő, szabályos lépéssorozat. Az eltáncolt rendezési algoritmusok csereorientáltak (ehhez lehetett kifejező tánclépéseket társítani): újra és újra összehasonlítanak, majd szükség esetén cserélnek, amíg a sorozat rendezett nem lesz.

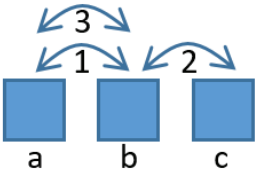
A rendezés természetesen általános feladat: tetszőleges számú érték növekvő sorrendbe rendezése. Most még csak három számmal dolgozunk, hogy a lépések jól követhetők legyenek, és közben azt is lássuk, hogy különböző algoritmusok ugyanazt a célt eltérő „ritmusban” érik el.

Tároljuk a három beolvasott számot az `a`, `b`, `c` változókban. A cél: `a <= b <= c`.

Egy rendezési algoritmus lényege, hogy bármilyen sorrendben legyenek is kezdetben az értékek, az algoritmus lefutása után az `a` változóban a legkisebb, a `b`-ben a középső, a `c`-ben pedig a legnagyobb érték álljon.

Buborékos rendezés (*bubble-sort*)

A buborékos rendezés ötlete: egymás melletti elemeket hasonlítunk össze, és ha rossz sorrendben vannak, felcseréljük őket. A nagyobb elemek fokozatosan „felfelé” vándorolnak (buborékoznak). Lásd az 5.1. ábrát.



5.1. ábra. *Buborékos rendezés három elemre: (1) hasonlítsd össze* `a` *és* `b` *értékét, szükség esetén cseréld fel őket; (2) hasonlítsd össze* `b` *és* `c` *értékét, szükség esetén cseréld fel őket; (3) hasonlítsd össze* `a` *és* `b` *értékét ismét, szükség esetén csere. Az első két lépés eredményeként a legnagyobb érték garantáltan a* `c` *változóba kerül. A harmadik lépés után a második legnagyobb érték a* `b`*-ben áll, így a legkisebb szükségképpen az* `a` *változóban marad.*

Az algoritmust megvalósító C kódrészlet:

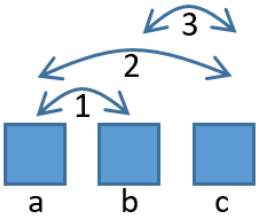
```
/* Buborékos rendezés 3 elemre */
if (a > b) { int v = a; a = b; b = v; }
if (b > c) { int v = b; b = c; c = v; }
if (a > b) { int v = a; a = b; b = v; }
```

Néhány észrevétel, amelyeket majd később kamatoztatunk igazán:

- ha az értékek már eleve növekvő sorrendben vannak, a három összehasonlítás során egyetlen csere sem történik (ez a legjobb eset – best case);
- ha kezdetben fordított sorrendben állnak, mindhárom összehasonlítást csere követi (ez a legrosszabb eset – worst case);
- ha az első két összehasonlítás során nem történik csere, akkor az értékek már rendezettek, és a harmadik ellenőrzés elhagyható (ennek beépítését a kódba az olvasóra bízzuk).

Kiválasztó rendezés (*selection-sort*)

A kiválasztó rendezés lényege: kiválasztjuk a legkisebb elemet, és az első helyre tesszük; majd a maradék elemek közül kiválasztjuk a következő legkisebbet. Lásd az 5.2. ábrát.



5.2. ábra. *Kiválasztó rendezés három elemre: (1) ha* `b` *kisebb, mint* `a`*, csere; (2) ha* `c` *kisebb, mint* `a`*, csere (most már az első helyen a legkisebb áll); (3) végül rendezni kell* `b` *és* `c` *viszonyát.*

A megfelelő C kódrészlet a következő:

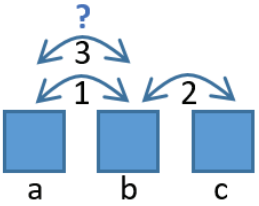
```
/* Kiválasztásos rendezés 3 elemre */
if (b < a) { int v = a; a = b; b = v; }
if (c < a) { int v = a; a = c; c = v; }
if (c < b) { int v = b; b = c; c = v; }
```

A „ritmus” itt eltér a buborékos rendezésétől: először a legkisebb elem kerül a helyére, majd a maradék két elem rendeződik.

A legjobb és legrosszabb esetre vonatkozó megjegyzések itt is érvényesek. A kiválasztásos rendezés sajátossága, hogy a helyes eredményhez minden esetben szükség van a harmadik összehasonlításra, és adott esetben cserére is.

Beszúró rendezés (*insertion-sort*)

A beszúró rendezés úgy működik, mintha kézzel rendezgetnénk kártyákat: a sorozat elején már rendezett rész található, és a következő elemet a megfelelő helyre „beszúrjuk”. Lásd az 5.3. ábrát.



5.3. ábra. *Beszúró rendezés három elemre: (1) először rendezzük* `a` *és* `b` *párosát; ezután* `c`*-t a helyére tesszük: (2) ha* `c` *kisebb, mint* `b`*, akkor* `b`*-t hátrébb kell tolni, (2) ha* `c` *kisebb, mint* `a`*, akkor* `a`*-t is hátrébb kell tolni. (A „hátrébb tolást” itt cserével oldjuk meg).*

Íme, a megfelelő C kódrészlet:

```
/* Beszúrásos rendezés 3 elemre */
if (a > b) { int v = a; a = b; b = v; }

if (c < b) {
    int v = c;
    c = b;
    b = v;

    if (b < a) { v = a; a = b; b = v; }
}
```

Itt a ritmus feltételesebb: ha `c` már eleve a végére való, akkor csak „átfut” a rendszer, és nincs szükség további beavatkozásra.

Vegyük észre, hogy mindhárom módszer ugyanazt a célt valósítja meg (rendezés), mégis eltérő lépéssorozatot követ. Ezek a különbségek nagy elemszám esetén válnak igazán fontossá, amire a későbbiekben még visszatérünk.



AI-csepp: indoklás

Kérdezd meg az AI-t: egy `n` elemű számsorozat (csereorientált) rendezésekor a buborékos, kiválasztó és beszúró rendezés hány összehasonlítást és hány cserét végez legjobb esetben és legrosszabb esetben?

- A válaszban különítsd el a három algoritmust, és röviden indokold is, mitől „legjobb” vagy „legrosszabb” egy bemenet (például már rendezett vagy fordított).

Tanulság: (1) nemcsak az számít, mit csinál egy algoritmus, hanem az is, hányszor teszi azt; (2) egy algoritmus viselkedése függ a bemenet jellegétől.

5.9. ECN szál: királyok a sakktáblán



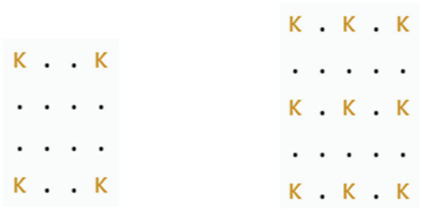
Az ECN-2023-as versenyen szerepelt egy olyan feladat, amely egy sakktáblára épülő elrendezési problémát vizsgált. A történet egyszerű volt, a mögötte álló gondolkodás azonban már algoritmikus szemléletet igényelt: miként lehet egy szabályrendszer mellett a lehető legtöbb elemet elhelyezni egy rácson.

A feladat mostani változatában egy $n \times n$ méretű sakktáblát tekintünk. A táblára királyokat szeretnénk elhelyezni úgy, hogy egyik király se támadhassa a másikat. Ismeretes, hogy a király minden irányban egy mezőt támad, ezért két király nem állhat sem közvetlen szomszédos mezőkön, sem átlósan egymás mellett.

Feladat: határozd meg, hogy legfeljebb hány király helyezhető el egy $n \times n$ méretű sakktáblán a fenti feltételek mellett!

A feladat célja nem az elrendezés konkrét megrajzolása, hanem annak felismerése, hogy a szabályrendszer rendszeres mintázatot kényszerít ki, és a maximális darabszám egy egyszerű képlettel leírható.

A megoldás során érdemes külön megvizsgálni az alábbi két esetet: amikor `n` páros, illetve amikor `n` páratlan. Nem nehéz átlátni, hogy `n` páros esetén egy sorban (illetve oszlopban) legfeljebb $n/2$ király helyezhető el úgy, hogy ne üssék egymást. Ez az érték $(n+1)/2$, amennyiben `n` páratlan. Lásd az 5.4. ábrát.



(a) (b)

5.4. ábra. *Maximális számú, egymást nem támadó királyok egy-egy helyes elhelyezése (a) 4×4-es és (b) 5×5-ös sakktáblán.*

Innen már csak egy tyúklépés választ el bennünket attól, hogy meghatározzuk a maximálisan elhelyezhető királyok számát a teljes sakktáblára vonatkozóan. A megfelelő képlet felismerése, valamint egy erre épülő, `if` utasítás köré szerveződő program megírása maradjon az olvasóra.

Ez a feladat szép példája annak, hogy egy lokális szabály (egy király mit támad) hogyan vezet globális maximumhoz.



AI-csepp: hányféleképpen érhető el a maximum?

Kérdezd meg az AI-t: egy $n \times n$ sakktáblán hány különböző módon helyezhető el maximális számú király úgy, hogy egyik se támadja a másikat páros, illetve páratlan `n` esetén?

- Egészítsd ki a kódodat ennek kiírásával is.

Tanulság: (1) a „maximális darabszám” nemcsak a mennyiségről szól, hanem arról is, mennyire kényszerített a szerkezet; (2) a tábla paritása (páros/páratlan n) alapvetően megváltoztatja, hogy egyértelmű-e a maximális elrendezés, vagy sokféle variáció lehetséges.

▶▶ Zárógondolat

Ebben a fejezetben megismerkedtünk az elágazás vezérlési szerkezetével, valamint az `if` utasítás szerepével. Láttuk, hogy a program végrehajtása nem

feltétlenül egyetlen, előre rögzített útvonalon halad: a feltételek kiértékelése alapján több lehetséges forgatókönyv közül választhatunk.

Az egyszerű kétutas elágazástól az egymásba ágyazott döntésekig ugyanaz az alapelv érvényesül: a program a futás közben szerzett információk alapján alkalmazkodik. Ez az alkalmazkodóképesség teszi lehetővé, hogy a kód ne csak egyetlen esetre, hanem egy egész esetcsoport kezelésére legyen alkalmas.

A következő fejezetben azt vizsgáljuk meg, miként léphetünk tovább: hogyan ismételhetünk műveleteket, és miként dolgozhatunk fel egész számsorozatokat a `for` ciklus segítségével.

6. Számsorozatok feldolgozása: ismert lépésszámú ismétlés

Az eddigi feladatokban legfeljebb néhány adatot kezeltünk egyszerre. Egy-egy számot, két változót, esetleg három együtthatót olvastunk be, és ezek alapján hoztunk döntéseket. A valós problémák jelentős részében azonban nem csupán néhány értékkel, hanem egész számsorozatokkal dolgozunk. Ezek az adatsorok származhatnak mérésekből, játékokból, statisztikákból vagy más adatgyűjtési folyamatokból.



Ebben a fejezetben arra összpontosítunk, hogy miként tudunk számsorozatokat feldolgozni. Megismerkedünk a `for` ciklus klasszikus változatával, valamint az egydimenziós tömbökkel mint tárolókkal.

6.1. A `for` ciklus – ismert lépésszámú ismétlés

A C nyelvben a `for` ciklus rendkívül rugalmas vezérlési szerkezet. Egyelőre a ciklus klasszikus változatával ismerkedünk meg, amely jól használható olyan feladatoknál, ahol az ismétlések száma előre ismert. A `for` ciklus teljes általánosságban vett használatát a következő fejezetben mutatjuk be.

6.1.1. Feldolgozás beolvasás közben

Az eddigi példáinkban mindig annyi változóra volt szükségünk, ahány adatot kellett eltárolnunk. A másodfokú egyenletet megoldó programunk a három együtthatót három változóban tárolta el (`a`, `b`, `c`), a barlangátkelés feladat esetében pedig a négytagú család átkelési idejének tárolásához négy változót használtunk (`a`, `b`, `c`, `d`).

Ez nagyon kézenfekvőnek tűnik, de vajon mindig kötelező ez? Vagy mindig ez a célszerű megoldás?

A másodfokú egyenlet esetében nincs más választásunk: a három együtthatóra egyszerre szükség van. Figyeljük meg azonban a barlangátkelés feladat egy leegyszerűsített részfeladatát, amikor a cél csupán az időértékek összegének meghatározása. Legyen tehát a feladat az, hogy olvassunk be négy számot, és számítsuk ki ezek összegét.

```
#include <stdio.h>

int main(){
    int osszeg = 0;
    int x;
```

```
    scanf("%i", &x); //beolvas x-be
    osszeg += x;     //hozzáadja x-et osszeg-hez

    scanf("%i", &x); //beolvas x-be
    osszeg += x;     //hozzáad x-et osszeg-hez

    scanf("%i", &x); //beolvas x-be
    osszeg += x;     //hozzáad x-et osszeg-hez

    scanf("%i", &x); //beolvas x-be
    osszeg += x;     //hozzáad x-et osszeg-hez

    printf("Osszeg = %i\n", osszeg);

    return 0;
}
```

Miben különbözik ez a megközelítés az előző fejezetben látott megoldástól? Ott először beolvastuk az input adatokat, majd feldolgoztuk őket. Itt a feldolgozás a beolvasással párhuzamosan történik, ezért nincs szükség az összes adat egyidejű eltárolására. Elég egyetlen változó (`x`), amely mindig az aktuálisan beolvasott értéket tartalmazza. A következő beolvasás felülírja ezt az értéket, de ez nem okoz adatvesztést, hiszen az adott szám már betöltötte a szerepét: hozzá lett adva az összeghez (`osszeg += x;`).

Ennek a megközelítésnek nemcsak az az előnye, hogy memóriatakarékosabb – hiszen mind a négy beolvasás ugyanabba a változóba történik –, hanem az is, hogy a beolvas-hozzáad műveletpáros minden esetben azonos.

Ez természetesen felvet egy kérdést. Ha a számítógépnek pontosan ugyanazt a műveletet kell többször elvégeznie, akkor ezt valóban csak az utasítások ismételt leírásával tehetjük meg? Vagy van mód arra, hogy egy adott utasítást, illetve utasítássorozatot – jelen esetben a beolvas-hozzáad párost – ismételten hajtassunk végre?

Éppen erre szolgálnak a ciklusutasítások.

Íme a megoldás `for` ciklus használatával.

```
#include <stdio.h>

int main(){
    int osszeg = 0;
    int x;

    for (int i = 1; i <= 4; i++) {
        scanf("%i", &x);
```

```
        osszeg += x;
    }

    printf("Osszeg = %i\n", osszeg);

    return 0;
}
```

A fenti kódban szereplő `for` ciklus utasítás szavakba öntve így fogalmazható meg: ismételd a ciklustörzsben szereplő utasításpárost minden `i`-re 1-től 4-ig. Ez gyakorlatilag azt jelenti, hogy a beolvas-hozzáad műveletpáros pontosan négyszer hajtódik végre.

A ciklusos megoldás nagy előnye – a tömörségén és átláthatóságán túl – az, hogy könnyen általánosítható tetszőleges elemszámú számsorozatra. Az alábbi kód a következő feladatot oldja meg: olvassuk be a be.txt állományból egy számsorozat elemszámát (n), majd az elemeit, és írjuk ki az elemek szorzatát a képernyőre.

```
#include <stdio.h>

int main(){
    int n;
    int x;
    int szorzat = 1;

    /* Bemenet átirányítása fájlból */
    if (freopen("be.txt", "r", stdin) == NULL) {
        printf("Nem sikerult megnyitni a be.txt fajlt.\n");
        return 1;
    }

    /* A sorozat elemszámának beolvasása */
    scanf("%i", &n);

    /* A sorozat elemeinek beolvasása és feldolgozása */
    for (int i = 1; i <= n; i++) {
        scanf("%i", &x);
        szorzat *= x;
    }

    printf("A szamsorozat elemeinek szorzata: %i\n", szorzat);

    return 0;
}
```

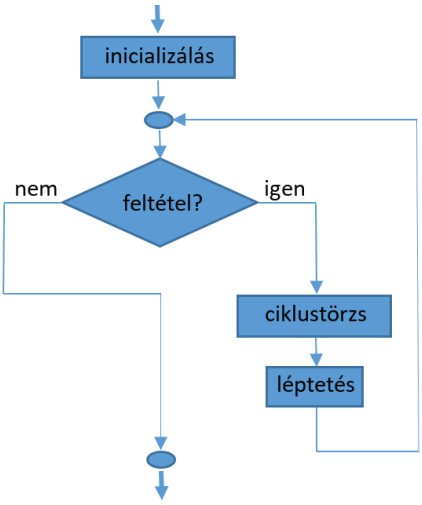
▦ **Algoritmikai mintázatok.** A fenti példák egyúttal bemutatták az *összeg- és szorzatszámítás technikáját* is. A módszer lényege, hogy egy segédváltozót kezdetben a megfelelő semleges elemre inicializálunk (összeg esetén 0-ra, szorzat esetén 1-re), majd a ciklus minden lépésében frissítjük (hozzáadunk, megszorozzuk) ezt az értéket az aktuálisan beolvasott szám felhasználásával.

</> Szintaxis. Itt az ideje, hogy megadjuk a `for` ciklus klasszikus változatának általános alakját.

```
for (<inicializálás>; <feltétel>; <léptetés>) {
    /* ciklustörzs */
}
```

A `for` ciklus három, pontosvesszőkkel elválasztott részből áll. A klasszikus változat használ egy ciklusváltozót, amely a fenti példákban az `i` volt.

- Inicializálás: a ciklus indulásakor egyetlen alkalommal hajtódik végre. Itt szokás megadni a ciklusváltozó kezdeti értékét (például `int i = 1`).
- Feltétel: a feltételt a program minden egyes ismétlés előtt kiértékeli. Amíg a feltétel igaz, a ciklus törzse végrehajtódik. Amint a feltétel hamissá válik, a ciklus befejeződik (például `i <= n`).
- Léptetés: a léptetés minden ciklus lefutás után hajtódik végre. Feladata a ciklusváltozó módosítása, tipikusan növelése vagy csökkentése (például `i++`).



6.1. ábra. *A `for` ciklus klasszikus változatának folyamatábrája*

A `for` ciklus működésének menete, lépésről lépésre:
- végrehajtódik az inicializálás,
- kiértékelődik a feltétel,
- ha igaz, lefut a ciklustörzs,
- végrehajtódik a léptetés,
- visszatérés a feltétel vizsgálatához.

Ez a folyamat addig ismétlődik, amíg a feltétel igaz. Ezt szemlélteti a 6.1. ábra.

[i] Megjegyzés. A `for` ciklus e klasszikus változata különösen alkalmas olyan feladatokra, ahol:
- előre ismert az ismétlések száma,
- számsorozat elemein szeretnénk végighaladni,
- vagy egy segédváltozót rendszeresen frissítünk.

6.1.2. Szomszédos elemek kezelése

Előfordul, hogy a feladat jellege megköveteli: ne csak az aktuális értéket ismerjük, hanem a számsorozat egy korábbi elemét is megőrizzük. Ilyenkor két (vagy több) változóra van szükség, amelyekben egymást követő elemeket tárolunk. Tipikus példa erre annak ellenőrzése, hogy egy számsorozat szigorúan növekvő-e. Az alábbi kód az első példa, amely a `for` és az `if` utasításokat kombinálja.

```
#include <stdio.h>

int main(){
    int n;
    scanf("%i", &n);

    int elozo, aktualis;
    int novekvo = 1;  /* feltesszük, hogy növekvő */

    /* Első elem beolvasása */
    scanf("%i", &elozo);

    /* A többi elemet mindig az előzővel hasonlítjuk */
    for (int i = 2; i <= n; i++) {
        scanf("%i", &aktualis);

        if (aktualis <= elozo) {
            novekvo = 0;
        }

        elozo = aktualis;  /* léptetjük az "elozo" értéket */
    }
```

```
    if (novekvo) {
        printf("A sorozat szigoruan novekvo.\n");
    } else {
        printf("A sorozat nem szigoruan novekvo.\n");
    }

    return 0;
}
```

Tömör magyarázat

- Kezdetben feltesszük, hogy a számsorozat növekvő (`novekvo = 1`), amíg az ellenkezője nem derül ki.
- Az első értéket beolvassuk az `elozo` változóba.
- Minden további elemet az `aktualis` változóba olvasunk be, és összehasonlítjuk az `elozo`-vel.
- Ha valaha `aktualis <= elozo`, akkor a sorozat nem szigorúan növekvő; átállítjuk a `novekvo` változó értékét nullára.
- Ezután az `elozo = aktualis` lépéssel „továbbcsúsztatjuk” az ablakot: a következő iterációban az eddigi aktuális lesz az előző.

▦ **Algoritmikai mintázat.** A fenti megoldás a *létezés-ellenőrzés technikáját* alkalmazza: azt vizsgáljuk, hogy létezik-e a sorozatban olyan elem (jelen esetben szomszédos elempár), amely megsérti az előírt feltételt. E módszer értelmében:

- először beállítunk egy segédváltozót egy „optimista” kezdeti értékre (például feltesszük, hogy a feltétel teljesül);
- ezután a sorozat feldolgozása közben, amint olyan elemet találunk, amely megsérti a feltételt, ezt a segédváltozót átállítjuk;
- a feldolgozás végén a segédváltozó értéke alapján döntjük el az eredményt.

Fontos, hogy az ilyen ellenőrzést megvalósító `if` utasításnak nem szabad `else` ága legyen: a segédváltozó visszaállítása hibás működéshez vezetne, mivel a feltétel megsértése után az eredmény már nem változhat.

A bemutatott megoldásban a feldolgozást a ciklus végéig folytatjuk, még akkor is, ha a feltétel megsértése már korábban kiderült. Ez a megoldás egyszerű és áttekinthető, ezért kezdésként jól használható. Általában azonban hatékonyabb megoldás érhető el, ha a feldolgozást azonnal megszakítjuk, amint az eredmény eldönthetővé válik. Ennek a korai leállásnak a technikáját – és az ehhez szükséges vezérlési eszközöket – később tárgyaljuk.

6.2. Egydimenziós tömbök

Eddig olyan feladatokat vizsgáltunk, ahol a számsorozat elemeit a beolvasással párhuzamosan dolgoztuk fel. Ezekben az esetekben elegendő volt egy vagy néhány változó használata, mivel az egyes adatokra a feldolgozás után már nem volt szükség.

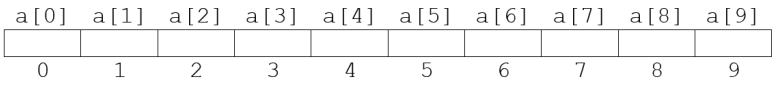
Vannak azonban olyan feladatok, ahol ez a megközelítés nem elegendő. Előfordulhat például, hogy a sorozat elemeire később újra szükség van, az adatokat többször is végig kell járni, az elemek sorrendjét módosítani kell, vagy az adatok között nemcsak szomszédos, hanem tetszőleges távolságú kapcsolatokat vizsgálunk.

Ilyen esetekben a számsorozatot egydimenziós tömbben eltároljuk, majd a feldolgozást ezen az eltárolt adatsoron végezzük el.

</> Szintaxis. A tömbváltozók többrekeszes változók, amelyek azonos típusú adatok tárolására alkalmasak, és amelyek egyes celláira indexeléssel hivatkozunk. Az egydimenziós tömbök deklarálásának általános alakja:

```
<típus> <név>[<méret>];
```

Ahogy a 6.2. ábráról is kiderül, a C nyelvben a tömbök indexelése automatikusan 0-tól indul. Az egyes cellákra – és az ott tárolt értékekre – a `<név>[<index>]` alakú tömbindexeléssel hivatkozunk.



6.2. ábra. *Szemléltető ábra az int a[10]; tömbdeklaráláshoz. A cellák alatt a megfelelő indexek szerepelnek, felettük pedig az egyes cellákra való hivatkozás módja, illetve a bennük tárolt értékek tömbindexelés révén történő elérése.*

Az alábbiakban egy példa látható egy számsorozat egydimenziós tömbbe történő beolvasására. A 6.3. ábra az ily módon feltöltött tömböt szemlélteti.

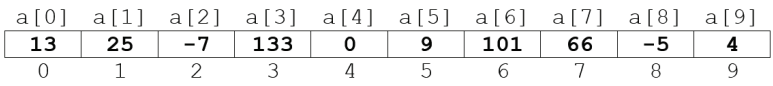
```
int n;
scanf("%i", &n);

int a[n];  /* deklarálunk egy n elemű int-tömböt */

/* Minden i-re 0-tól (n-1)-ig */
for (int i = 0; i < n; i++) {
```

```
    scanf("%i", &a[i]);
}
```

A fenti kódrészletben a teljes számsorozat az `a` tömbbe kerül eltárolásra. Kihasználtuk, hogy a `for` ciklus ciklusváltozója sorban felveszi a 0, 1, 2, … értékeket, amelyeket a tömb celláinak indexelésére használunk. Ezt követően a tömb elemei tetszőleges alkalommal és sorrendben feldolgozhatók.



6.3. ábra. *Szemléltető ábra arra az esetre vonatkozóan, amikor egy 10 elemű tömbbe a {13, 25, -7, 133, 0, 9, 101, 66, -5, 4} tízelemű számsorozatot olvassuk be.*

[i] Megjegyzés. Tömbök esetében is van lehetőség deklaráláskor kezdőértéket adni az elemeknek. Például:

```
int a[5] = {10, 20, 30, 40, 50};
```

Ez azt jelenti, hogy az `a[0]` értéke `10`, az `a[1]` értéke `20`, és így tovább. Fontos, hogy ez kezdőértékadás, vagyis a tömb létrehozásakor történik. Később a teljes tömbre már nem adhatunk értéket egyetlen értékadással; az elemeket külön-külön vagy ciklussal módosíthatjuk.

6.2.1. Feldolgozás eltárolt adatokon

Az eltárolás lehetővé teszi, hogy a feldolgozás és a beolvasás időben elkülönüljön. Az alábbi kódrészlet kiszámítja és kiírja az eltárolt számsorozat páros (2-vel osztva 0 maradékú), illetve páratlan (2-vel osztva 1 maradékú) elemeinek átlagát.

▦ **Algoritmikai mintázatok.** Az *átlagszámítás technikája* az összegzés és a megszámlálás kombinációja. Az összegzés technikáját már korábban bemutattuk. A *számlálás technikája* során egy nulláról induló számláló változó segítségével, annak léptetésével (++) határozzuk meg, hogy a számsorozat hány eleme felel meg egy adott feltételnek. Az átlagszámításhoz a feldolgozás során párhuzamosan gyűjtjük az elemek összegét és darabszámát, majd az átlagot csak a ciklus befejezése után számítjuk ki az összeg és a darabszám hányadosaként. Az osztásnál ügyelni kell arra, hogy a darabszám ne legyen nulla, valamint arra is, hogy lebegőpontos osztás történjen, különben egész osztás miatt pontatlan eredményt kapunk.

```
int osszeg1 = 0, osszeg2 = 0;
int db1 = 0, db2 = 0;

for (int i = 0; i < n; i++) {
    if (a[i] % 2 == 1) {
        osszeg1 += a[i];
        ++db1;
    } else {
        osszeg2 += a[i];
        ++db2;
    }
}

if (db1 > 0) {
    printf("A paratlanok atlaga: %f\n", (float)osszeg1 / db1);
} else {
    printf("Nincs paratlan szam a sorozatban.\n");
}

if (db2 > 0) {
    printf("A parosak atlaga: %f\n", (float)osszeg2 / db2);
} else {
    printf("Nincs paros szam a sorozatban.\n");
}
```

Ebben az esetben a feldolgozás már nem függ a beolvasás menetétől; a tömb elemein önállóan járunk végig.

Az önálló végigjárás azt is jelentheti, hogy a feldolgozást fordított sorrendben végezzük. A következő kódrészlet a számsorozat elemeit fordított sorrendben írja ki.

```
/* Minden i-re (n-1)-tól 0-ig */
for (int i = n-1; i >= 0; i--) {
    printf("%i ",a[i]);
}
```

[i] Megjegyzés. Az eltárolt számsorozaton végzett feldolgozás nagyobb rugalmasságot biztosít, ugyanakkor több memóriát igényel. A programozónak minden feladatnál mérlegelnie kell, hogy elegendő-e a beolvasással párhuzamos feldolgozás, vagy szükség van a teljes adatsor eltárolására. Ez a döntés – hasonlóan a korábbiakhoz – algoritmikus jellegű, és jelentősen befolyásolja a megoldás szerkezetét.

6.2.2. Számsorozatok generálása és eltárolása

Az eddigi példákban a számsorozatok elemei külső forrásból (billentyűzetről vagy fájlból) származtak. Gyakran azonban nincs szükség külső bemenetre: a számsorozat elemei egy szabály vagy képlet alapján, illetve véletlenszerűen is előállíthatók.

Ilyen esetekben a program maga generálja az adatokat. Amennyiben szükséges a számsorozat eltárolása, a tömb feltöltése ilyenkor is tipikusan `for` ciklus segítségével történik, ahol a ciklusváltozó határozza meg, hogy éppen a tömb melyik cellájába kerül az adott érték.

Az alábbi kódrészlet képlet alapján állít elő (és tárol el tömbben) egy számsorozatot oly módon, hogy az egyes elemek értéke az indexük négyzetével egyenlő.

```
for (int i = 0; i < n; i++) {
    a[i] = i * i;
}
```

Ebben az esetben a számsorozat elemei nem bemenetből származnak, hanem egy egyszerű szabály alapján keletkeznek: a ciklusváltozó (`i`) egyszerre szolgál indexként és a képlet változójaként.

Az alábbi `for` ciklus a Fibonacci sorozat elemeivel tölti fel a `fibo` tömböt: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

```
fibo[0] = 0;
fibo[1] = 1;
for (int i = 2; i <= n; i++) {
    fibo[i] = fibo[i-1] + fibo[i-2];
}
```

Az is előfordulhat, hogy a számsorozat elemeit véletlenszerűen szeretnénk előállítani, például tesztelés vagy szimuláció céljából. Az alábbi példában 0 és 99 közötti egész számokat generálunk és tárolunk el tömbben:

```
#include <stdlib.h>

for (int i = 0; i < n; i++) {
    a[i] = rand() % 100;
}
```

Fontos megjegyezni, hogy a `rand()` függvény nem valódi véletlen számokat állít elő, hanem pszeudovéletlen sorozatot. Ez abból is nyilvánvaló, hogy ha nem inicializáljuk a véletlenszám-generátort, akkor a program minden futtatásakor

ugyanazt a sorozatot kapjuk. A generátor inicializálására a `srand()` függvény szolgál.

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

srand(time(NULL));

for (int i = 0; i < n; i++) {
    a[i] = rand() % 100;
}
```

A time(`NULL`) az aktuális időt adja vissza, amelyet gyakran használunk a srand() függvény paramétereként, hogy a generált pszeudovéletlen számsorozat minden futtatáskor eltérő legyen.

▶ Összegzés

Az alábbiakban összefoglaljuk a fejezetben tárgyalt tipikus számsorozat-kezelési eseteket:

• feldolgozás beolvasás közben

```
for (int i = 1; i <= n; i++) {
    scanf("%d", &x);
    /* feldolgozzuk x-et */
}
```

• szomszédos elemek vizsgálata

```
scanf("%d", &elozo);
for (int i = 2; i <= n; i++) {
    scanf("%d", &aktualis);
    /* vizsgáljuk az előző, aktuális párost */
    elozo = aktualis;
}
```

• számsorozat eltárolása tömbben

```
for (int i = 0; i < n; i++) {
    scanf("%d", &a[i]);
}
/* a tömb újra bejárasával feldolgozzuk a számsorozatot */
```

• generált számsorozat előállítása

```
for (int i = 0; i < n; i++) {
    a[i] = rand() % 100;
}
```

A számsorozatokkal kapcsolatos feladatok megoldásának első lépése mindig annak eldöntése, hogy a sorozat elemeit hogyan kezeljük. Ez a döntés határozza meg, hogy szükség van-e tömbre, hány változót használjunk a tároláshoz, és milyen szerkezetű `for` ciklust alkalmazzunk.



AI-csepp – tipikus hibák
Az alábbi lista a fejezet anyagához kapcsolódó leggyakoribb hibákat foglalja össze.

Tipikus `for` ciklus hibák:
• Rosszul megválasztott ciklushatárok (`i <= n` vs. `i < n`)
• A 0-tól induló indexelés és az 1-től induló számlálás keverése
• Fordított bejárásnál hibás ciklusfeltétel
Tömbök és indexelés tipikus hibái:
• Túlindexelés (`a[n]` használata az utolsó elem helyett)
• A szerepek összekeverése: a ciklusváltozó az index, az `a[i]` pedig érték
• Nem inicializált tömb vagy tömbelem használata
Algoritmikus gondolkodás – „mintázathibák”:
• A semleges elem hibás megválasztása (összegzésnél, szorzásnál stb.)
• Létezés-ellenőrzésnél hibás `else` ág alkalmazása
• Szomszédos elemek vizsgálatánál az első elem kezelése vagy az „ablakcsúsztatás” elmaradása

Kérdés az AI-hoz: Kérd meg, hogy fejtse ki, milyen tanulságok vonhatók le a fenti hibakategóriákból, és hogyan kerülhetők el ezek a hibák a gyakorlatban.

6.3. BBC szál: csoki vagy csili 3.



Végre eleget tudunk ahhoz, hogy megoldjuk a csoki vagy csili feladatot tetszőleges cukorkaszámra. Adott *n*, a cukorkák száma; írjunk programot, amely lehetővé teszi, hogy a számítógép a felhasználó ellen játsszon a „kiegészítés 4-re” stratégia szerint (lásd a 3.7. alfejezetet).

A 3.7. és 5.7. alfejezetekben tisztáztuk, hogy a gép a „nulladik lépésben” `n % 4` cukorkát vesz el; ha ez 0, akkor átengedi a kezdést az ellenfélnek. Ezt követően

minden fordulóban összesen 4 cukorka tűnik el, ezért a teljes „felhasználó + gép” fordulók száma *n* / 4, ami ismert lépésszámú ismétlés – a `for` ciklus tehát ideális. Az alábbi kód értelmezését az olvasóra bízzuk.

```
#include <stdio.h>

int main(){
    int n;
    scanf("%i", &n);

    int marad = n;  // marad: a cukorkák aktuális száma
    int r = n % 4;  // ennyit vesz el a gép kezdésként

    /* A gép "nulladik lépése" */
    if (r != 0) {
        printf("Gep: elvesz %i cukorkat.\n", r);
        marad -= r;
    } else {
        printf("Te kezdesz.\n");
    }

    /* Ezután minden fordulóban 4 cukorka tűnik el. */
    int fordulok = marad / 4;
    int jatekos, gep;

    for (int f = 1; f <= fordulok; f++) {

        /* Játékos lépése: 1..3 cukorka */
        printf("\n%i. fordulo - Marad: %i\n", f, marad);
        printf("Te mennyit veszel el? (1-3): ");
        scanf("%i", &jatekos);
        marad -= jatekos;

        /* Gép lépése: kiegeszítés 4-re */
        gep = 4 - jatekos;
        printf("Gep elvesz %i cukorkat.\n", gep);
        marad -= gep;
    }

    printf("Marad: %i\n", marad);
    printf("A gep vette el az utolsot. A gep nyert.\n");

    return 0;
}
```

Ellenőrizd a megértésedet az alábbi feladatok megoldásával.

- Indokold meg, hogy a program miért mindig a gépet hirdeti győztesnek, függetlenül a felhasználó tényleges lépéseitől.
- Általánosítás. Adott n (a cukorkák száma) és k (egy 2-nél nagyobb egész szám). Minden fordulóban a játékosok 1..($k-1$) darab cukorkát vehetnek el. Módosítsd a kódot úgy, hogy a számítógép a „kiegészítés k-ra” stratégia szerint játsszon.



AI-csepp

Add meg az AI-nak a feladat leírását és a fenti kódot, majd kérd meg, hogy véleményezze azt.

- A kapott válasz alapján fogalmazd meg, mit tanultál a program működéséről, illetve a stratégia és a megvalósítás kapcsolatáról.

6.4. AlgoRythmics szál: lineáris keresés



Az AlgoRythmics kollekció flamenco tánccal illusztrálja a lineáris keresés algoritmusát. A koreográfia története szerint egy fiú a párját keresi a lányok sorában: azt a lányt, aki a hátán ugyanazt a számot viseli, mint ő. Mivel a fiú szemben áll a lányokkal, nem látja, ki a párja. Ezért kénytelen felkérni a lányokat egy-egy táncra, hogy kiderüljön, ugyanazt a koreográfiát táncolják-e, azaz ugyanazt a számot képviselik-e, mint ő.

▦ **Algoritmikai mintázat.** A fiú *lineáris keresési* stratégiát alkalmaz: sorban halad végig a lányokon, és egyesével ellenőrzi, hogy megtalálta-e a keresett párt. Ha igen, megáll; ha pedig a sor végére ér, a keresés sikertelen.

Mivel a lányok szemben, a fiú viszont háttal van a nézőnek, a fiú hátán a néző látja a keresett számot, a lányok által viselt számok – vagyis az a számsorozat, amelyben a keresés zajlik – azonban a néző számára rejtettek (lásd a 6.4. ábrát). Ez a nézőpont jól modellezi azt a helyzetet, hogy az algoritmust a „vak számítógépnek” kell egyértelmű, lépésről lépésre követhető utasításokként megadni. Ahogyan a fiúnak sorra fel kell kérnie a lányokat egy-egy táncra, ugyanúgy a számítógép is csak egy összehasonlítási művelet révén tudja eldönteni, hogy a számsorozat aktuális eleme megegyezik-e a keresett számmal.



6.4. ábra. *Lineáris keresési algoritmus flamenco tánccal. (a) A székek, amelyeken a lányok ülnek, az egydimenziós tömb celláit képviselik. (b) A lányok az eltárolt számsorozat elemeit ábrázolják, a fiú pedig a keresett értéket. (c) A fiú egymás után kéri fel a lányokat, azaz lineáris keresést végez. (d) A fiú az x[6] széken találja meg a párját: a 7-es számot viselő lányt, aki az ő tükörkoreográfiáját táncolja vele teljes harmóniában.*

A szóban forgó keresési probléma a programozás nyelvén a következőképpen fogalmazható meg:

- Legyen egy `n` elemű sorozat, amelyet tipikusan egy tömbben tárolunk (`x[0..n-1]`), valamint egy keresett érték `k`.
- Feladat: döntsük el, hogy `k` szerepel-e a sorozatban; ha igen, adjuk meg egy előfordulásának indexét (pozícióját), ha nem, jelezzük a sikertelenséget.

Az alábbiakban a lineáris keresési algoritmus egy lehetséges implementációja látható. A keresés a sorozat elejétől indul, és egymás után hasonlítja össze az elemeket a keresett értékkel. Találat esetén megjegyzi az adott elem indexét; találat hiányában a `pozicio` változó értéke az előre beállított kezdőérték (-1) marad.

```
int pozicio = -1;   // -1 jelzi, hogy nincs találat
```

```
for (int i = 0; i < n; i++) {
    if (x[i] == k) {
        pozicio = i;   // eltároljuk az aktuális találat helyét
    }
}

if (pozicio != -1) {
    printf("Talalat: %i, index = %i\n", k, pozicio);
} else {
    printf("Nincs talalat!\n");
}
```

A fenti implementáció több előfordulás esetén a keresett érték utolsó előfordulásának pozícióját adja meg. Amennyiben az eredményt nem 0-tól, hanem 1-től indexelve szeretnénk megjeleníteni, a kiírandó érték `pozicio + 1`.

Ha az utolsó előfordulás helye érdekes számunkra, akkor a fenti megoldás megfelelő lehet. Amennyiben azonban csupán arra vagyunk kíváncsiak, hogy a keresett érték szerepel-e a sorozatban, illetve ha igen, akkor melyik az első előfordulási helye, ez a megoldás nem tekinthető hatékonynak, mivel a keresés nem áll le az első találat után.

Az alábbi módosított `for` ciklus ezt a problémát orvosolja. Egyúttal azt is szemlélteti, hogy a `for` ciklus feltétele a két pontosvessző között tetszőleges logikai kifejezés lehet. Az `&&` operátor lehetővé teszi a ciklus korai leállítását: amint találat történik, a `pozicio` változó értéke már nem `-1`, a ciklusfeltétel hamissá válik, és a ciklus befejeződik.

```
for (int i = 0; i < n && pozicio == -1; i++) {
    if (x[i] == k) {
        pozicio = i;   // első találat eltárolása
    }
}
```

Időbonyolultság

Ahogyan arra korábban már utaltunk, egy algoritmus vizsgálata feltételezi annak elemzését is, hogy miként működik legjobb, illetve – különösen fontos módon – legrosszabb esetben. Keresési algoritmusok esetén a legjobb eset akkor következik be, amikor a keresett érték már a sorozat első megvizsgált elemével megegyezik, míg a legrosszabb esetben a keresett érték egyáltalán nem szerepel a sorozatban, és ez csak a keresés végére derül ki.

A lineáris keresés esetében a legjobb esetben mindössze egy összehasonlításra van szükség, míg a legrosszabb esetben a sorozat összes elemét meg kell vizsgálni. Ennek megfelelően a végrehajtási idő – amely kéz a kézben jár az elvégzendő

alapműveletek számával (jelen esetben az összehasonlítások számával) – a sorozat elemszámával arányosan növekszik: `n` hosszú sorozat esetében `n` összehasonlításra van szükség. Ezt formálisan úgy írjuk le, hogy az algoritmus időbonyolultsága `O(n)`, azaz lineáris függvény szerint növekszik. Ez az összefüggés magyarázza az algoritmus elnevezését is: a keresés ideje lineárisan függ a bemenet méretétől, ezért nevezzük ezt az eljárást lineáris keresésnek.

Íme, néhány kapcsolódó kérdés, amelyen érdemes elgondolkodni:

- Hogyan határozhatnánk meg hatékonyan a keresett érték utolsó előfordulásának helyét?
- Miként módosulna a megoldás, ha a keresett érték összes előfordulásának pozícióira vagyunk kíváncsiak?
- Adj példát a lineáris keresés szempontjából legjobb és legrosszabb esetnek számító bemenetre (`n`, `x[0..n-1]`, `k`)!
- Ha a számsorozat elemeit tetszőleges sorrendben vizsgálnánk, javulhatna-e az algoritmus időbonyolultsága?
-



AI-csepp: reflexió
Kérdezd meg az AI-t: milyen gyakorlati helyzetekben indokolt a lineáris keresés alkalmazása, és milyen feltételek mellett tekinthető hatékony megoldásnak?

6.5. Hiúz szál: minimum és maximum keresés tetszőleges elemszámú sorozatban



Általánosítsuk a barlangátkelés feladatot (3.8. és 5.8. alfejezetek) egy *n* tagú családra. A barlang továbbra is kétszemélyes, a családnak csak egy elemlámpája van, és a családtagok átjutási ideje tetszőleges sorrendben adott. Továbbra is „a leggyorsabb futárkodik” stratégiában vagyunk érdekeltek.

A feladat megoldása feltételezi egy `n` elemű számsorozat legkisebb elemének meghatározását. Ez a probléma a lineáris keresés egy sajátos változatának tekinthető: nem egy előre megadott értéket keresünk, hanem a sorozat legkisebb vagy legnagyobb elemét határozzuk meg.

▦ **Algoritmikai mintázat.** A *minumum- és maximumkeresési* algoritmus kezdetben a sorozat első elemét tekinti aktuális minimumnak (maximumnak). Ezután sorban végighalad a további elemeken, és minden lépésben összehasonlítja az aktuális elemet az eddig ismert minimum (maximum) értékkel. Amennyiben ennél kisebb (nagyobb) elemet talál, frissíti az aktuális minimum (maximum) értékét. A sorozat végére érve az aktuálisan tárolt érték a keresett minimum (maximum).

Az alábbi implementáció a tomb[0..n-1] számsorozatban keresi meg a legkisebb és legnagyobb értékeket:

```
int min = tomb[0];
int max = tomb[0];

for (int i = 1; i < n; i++) {
    if (tomb[i] < min) {
        min = tomb[i];
    }
    if (tomb[i] > max) {
        max = tomb[i];
    }
}

/* a ciklus végén:
   min – a legkisebb elem értéke
   max – a legnagyobb elem értéke */
```

A minimum- és maximumkeresés során a sorozat minden elemét pontosan egyszer kell megvizsgálni, ezért az algoritmus időbonyolultsága $O(n)$, függetlenül attól, hogy a minimum vagy maximum érték a sorozat elején, végén vagy közepén található.

A megértés elmélyítését a következő feladatok szolgálhatják:

- Hogyan módosul az algoritmus, ha a minimum/maximum pozíciójára (indexére) is kíváncsiak vagyunk? Miért tekinthető a pozíció több információnak, mint pusztán az érték?
- Hogyan változik a megoldás, ha a minimum/maximum összes előfordulásának pozícióit szeretnénk meghatározni?
- Miben különböznek azok az algoritmusok, amelyek több minimum-/maximum-előfordulás esetén az első, illetve az utolsó előfordulás helyét adják meg?
- Hogyan keresnéd meg egy sorozat két legkisebb, illetve két legnagyobb elemét?
- Hogyan általánosítható a megoldás három legkisebb/legnagyobb elemre?
- Hogyan keresnéd meg adott tulajdonságú elemek közül a legkisebbet/legnagyobbat? (Például: a legkisebb páros vagy a legkisebb pozitív érték a sorozatban.)
- Extra: Implementáld a barlangátkelés feladat megoldását (a leggyorsabb futárkodik stratégiát) *n*-tagú családra.



AI-csepp

A minimum- és maximumkeresés egy másik lehetséges megvalósítása az, hogy a min/max változót nem a sorozat első elemével, hanem egy megfelelő, „sorozaton kívüli” kezdőértékkel inicializáljuk. Ez a megoldás bizonyos esetekben egyszerűbbnek tűnhet, ugyanakkor rejtett feltételezéseket és kockázatokat is hordozhat.

Kérd meg az AI-t, hogy magyarázd el ezt a megközelítést, és hogy térjen ki az előnyeire, hátrányaira, valamint azokra az esetekre, amikor ez a megoldás előnyös, illetve amikor veszélyes vagy hibás lehet.

6.6. ECN szál: kétszemélyes játék

ECN

Évekkel ezelőtt, a CEOI informatika versenyen (CEOI 2026) szerepelt egy olyan feladat, amelyben adott egy n elemű, pozitív egész számokból álló sorozat, ahol n páros és $n \geq 2$. A feladat egy kétszemélyes játék megvalósításából állt, amelyben a számítógép a felhasználó ellen játszik a következő egyszerű szabály szerint: (i) minden lépésben az aktuális játékos elvesz egy számot a sorozat valamelyik végéről (balról vagy jobbról); (ii) a játék végén az a játékos nyer, aki nagyobb összértéket gyűjtött össze. (Lásd a 6.5. ábrát.)

19	2	4	16	3	15	4	14	17	1
1	2	3	4	5	6	7	8	9	10

6.5. ábra. *Példa számsorozat n = 10 esetén. Mivel a játékosok felváltva választanak elemet a számsorozat valamelyik végéről (bal/jobb), a játék n/2=5 fordulós lesz, és mindketten öt-öt elemet tudhatnak majd magukénak.*

A kihívás abban állt, hogy a versenyzők találjanak egy garantáltan nyerő stratégiát a kezdő játékos számára (azaz legalább döntetlent biztosítót). Ezt a stratégiát kellett implementálni a számítógép mint kezdő játékos számára.

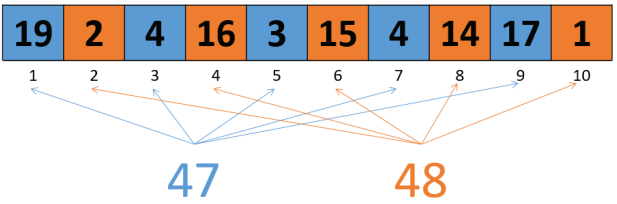
E feladat rávilágít arra, hogy egy probléma megoldása sokszor egyetlen kreatív felismerést igényel.

A kulcsészrevétel abban áll, hogy a kezdő játékos képes biztosítani, hogy vagy az összes páratlan indexű, vagy az összes páros indexű elemet gyűjtse össze. Ezt követően a feladat lényegében arra redukálódik, hogy megvizsgáljuk, a páros vagy a páratlan pozíciókban található elemek összege a nagyobb, és a gép ennek megfelelően választja meg a lépéseit. Ezt a stratégiát szemlélteti a 6.6. ábra.

Az alábbi implementációból az is kiderül, hogy a gép számára valójában csak az első lépés bír jelentőséggel, ezzel „üti meg a hangot”. Ezért tárgyaljuk külön az

első fordulót. Ezt követően mindig ugyanarról az oldalról választ, mint az ellenfele: ha az balról vett el elemet, akkor ő is balról, ha pedig jobbról, akkor ő is jobbról.

Ennek magyarázata abban rejlik, hogy mivel a páros és páratlan indexű elemek váltakozva helyezkednek el a számsorozatban, az ellenfél választása után az azonos végponton elérhető szomszédos elem mindig ellentétes paritású lesz. Így a kezdő játékos következetesen az általa preferált (páros vagy páratlan) indexű elemeket tudja begyűjteni.



6.6. ábra. *Mivel a páros pozíciókban található elemek összege nagyobb, mint a páratlan pozíciókban lévőké, a játék menete a következő: a kezdő játékos az első lépésben választhat egy páratlan vagy egy páros indexű elem között (az 1. vagy a 10. elem). Ha a páros indexű elemeket preferálja, és így a 10. elemet választja, akkor az ellenfélnek két páratlan indexű elem közül kell választania (az 1. vagy a 9. elem). Ez a lépés egy újabb páros indexű elemet szabadít fel a kezdő játékos számára, és a folyamat hasonló módon folytatódik, míg az összes páros indexű elemet a kezdő játékos, a páratlan indexűeket pedig az ellenfele gyűjti be.*

A „JÁTÉK ELŐKÉSZÍTÉSE” kódszakasz megírását az olvasóra hagyjuk. Előlegezzük, hogy az `n` változó tárolja a számsorozat hosszát, az `a[0..n-1]` tömb a számsorozat elemeit, míg a `paros_osszeg` és `paratlan_osszeg` változók rendre a páros, illetve páratlan indexű elemek összegét tartalmazzák.

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

int main() {
    /* A JÁTÉK ELŐKÉSZÍTÉSE */

    // A gép eldönti, melyik paritásra hajt
    // 0 = páros indexű elemek
    // 1 = páratlan indexű elemek
    int gep_erre_hajt;
    if (paros_osszeg >= paratlan_osszeg) {
        gep_erre_hajt = 0;
    } else {
```

```
    gep_erre_hajt = 1;
}

int bal_index = 0;
int jobb_index = n - 1;

int gep_pont = 0;
int jatekos_pont = 0;

int jatekos_valasztas;

/* ===== 1. FORDULÓ ===== */
printf("---- 1. fordulo ----\n");

// Gép első lépése (stratégiai)
if ((bal_index % 2) == gep_erre_hajt) {
    gep_pont += a[bal_index];
    printf("Gep balrol vesz: %i\n", a[bal_index]);
    bal_index++;
} else {
    gep_pont += a[jobb_index];
    printf("Gep jobb vesz: %i\n", a[jobb_index]);
    jobb_index--;
}

// Játékos első lépése
printf("Valassz (1 = bal, 2 = jobb): ");
scanf("%i", &jatekos_valasztas);

if (jatekos_valasztas == 1) {
    jatekos_pont += a[bal_index];
    printf("Jatekos balrol vett: %i\n\n", a[bal_index]);
    bal_index++;
} else {
    jatekos_pont += a[jobb_index];
    printf("Jatekos jobbrol vett: %i\n\n", a[jobb_index]);
    jobb_index--;
}

/* ===== TOVÁBBI FORDULÓK ===== */
for (int fordulo = 2; fordulo <= n / 2; fordulo++) {
    printf("---- %i. fordulo ----\n", fordulo);

    // Gép a játékos előző fordulóbeli választását követi
    if (jatekos_valasztas == 1) {
```

```
            gep_pont += a[bal_index];
            printf("Gep balrol vesz: %i\n", a[bal_index]);
            bal_index++;
        } else {
            gep_pont += a[jobb_index];
            printf("Gep jobbrol vesz: %i\n", a[jobb_index]);
            jobb_index--;
        }

        // Játékos aktuális fordulóbeli lépése
        printf("Valassz (1 = bal, 2 = jobb): ");
        scanf("%i", &jatekos_valasztas);

        if (jatekos_valasztas == 1) {
            jatekos_pont += a[bal_index];
            printf("Jatekos balrol vett: %i\n\n", a[bal_index]);
            bal_index++;
        } else {
            jatekos_pont += a[jobb_index];
            printf("Jatekos jobbrol vett: %i\n\n",a[jobb_index]);
            jobb_index--;
        }
    }

    printf("Gep pontszama: %i\n", gep_pont);
    printf("Jatekos pontszama: %i\n", jatekos_pont);

    return 0;
}
```



AI-csepp

Kérdezd meg az AI-t: a fenti megoldásban több ismétlődő, szinte teljesen azonos kódrészlet is megfigyelhető; elkerülhető-e ez az ismétlés, és ha igen, hogyan?

• A válasz minden bizonnyal túl fog mutatni az eddig tanult eszközökön, azonban már most érdemes elgondolkodni rajta. Az ilyen kérdések előrevetítik a későbbi fejezetek témáit, és kíváncsiságot ébresztenek arra, miként lehet egy programot rövidebbé, áttekinthetőbbé és karbantarthatóbbá tenni.

►► Zárógondolat

Ebben a fejezetben a számsorozatok feldolgozását szorosan összekapcsoltuk a `for` ciklus klasszikus változatával. Ez nem véletlen: az előre ismert lépésszámú ismétlés és a sorozatok elemein való sorban haladás természetes módon illeszkedik

ehhez a vezérlési szerkezethez. Ugyanakkor fontos hangsúlyozni, hogy ez a kapcsolat nem kizárólagos. A számsorozatok feldolgozása nem csak `for` ciklussal valósítható meg, és a `for` ciklus sem csupán erre a célra szolgál. A következő fejezetekben látni fogjuk, hogy a `while` és `do-while` ciklusok, valamint a `for` ciklus általánosabb használata más problématípusokra és gondolkodásmódokra is lehetőséget ad.

A fejezet során több alapvető algoritmikai mintázattal is megismerkedtünk: összegzés, szorzatszámítás, megszámlálás, létezés-ellenőrzés, szomszédos elemek vizsgálata, lineáris keresés, minimum- és maximumkeresés. Ezek a mintázatok nem önálló „trükkök”, hanem olyan újra és újra visszatérő gondolkodási sémák, amelyekre a későbbi, összetettebb algoritmusok is épülnek. A most megszerzett alaptudás ezért nem lezárt egység, hanem kiindulópont: eszköztár, amelyre a következő fejezetek vezérlési szerkezetei és algoritmusai tudatosan támaszkodnak majd.

A továbblépés kulcsa annak felismerése, hogy a ciklusok és a számsorozatok kezelése mögött mindig egy világos algoritmikus döntés áll: mit tárolunk el, mit dolgozunk fel azonnal, mikor és milyen feltétellel ismétlünk. Ezekre a kérdésekre a következő fejezetek új nézőpontokat és megoldási lehetőségeket kínálnak.

7. Amíg igaz: feltételek által vezérelt ismétlés

Az előző fejezetben olyan problémákat vizsgáltunk, ahol az ismétlések száma előre ismert volt: tudtuk, hány elemből áll a számsorozat, hányszor kell végrehajtani egy művelet, és ebből adódott az is, hogy meddig tart a feldolgozás. Ilyen esetekben természetes választás a `for` ciklus klasszikus változata. A valós problémák jelentős részében azonban nem tudjuk előre, hányszor kell ismételnünk egy utasítássorozatot. Előfordulhat, hogy addig olvasunk adatokat, amíg egy bizonyos feltétel teljesül, amíg hiba nem jelentkezik, vagy amíg a keresett eredményt meg nem találjuk. Ilyenkor nem az az elsődleges kérdés, hogy hányszor, hanem az, hogy meddig ismétlünk.



Ebben a fejezetben elsősorban olyan helyzeteket vizsgálunk, amelyeket a feltételek által vezérelt ciklusok írnak le jól. Ezek közül a `while` és a `do-while` ciklusokkal, valamint a `for` ciklus általánosabb használatával ismerkedünk meg.

7.1. A ciklusutasítások általános alakja

</> Szintaxis. Mielőtt konkrét példákon keresztül vizsgálnánk a különböző ciklusokat, tekintsük át azok általános alakját. Ezek a sémák megmutatják, hogy a ciklusok milyen szerkezeti elemekből épülnek fel, de önmagukban még nem árulják el, mikor melyiket érdemes használni.

```
while (<feltétel>) {
    /* ciklustörzs */
}
```

`while` mintázat: amíg a feltétel igaz, ismételd a ciklustörzset.

```
do {
    /* ciklustörzs */
} while (<feltétel>);
```

`do-while` mintázat: ismételd a ciklustörzset, amíg a feltétel igaz.

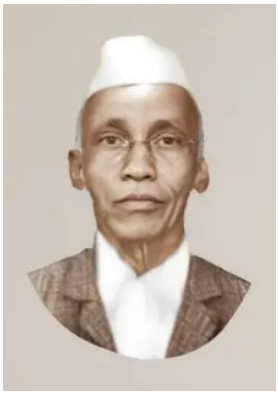
```
for (<inicializálás>; <feltétel>; <léptetés>) {
    /* ciklustörzs */
}
```

`for` mintázat: végezd el az inicializálást, majd amíg a feltétel igaz, ismételd a ciklustörzs+léptetés párost.

Mindhárom ciklusszerkezet esetében egy feltétel dönti el, hogy a ciklus folytatódik-e vagy befejeződik. A különbség abban rejlik, hogy mikor és hogyan kerül sor a feltétel kiértékelésére, illetve hogy a ciklus vezérlése mennyire van egy helyre koncentrálva.

Bár a C nyelvben a `for`, `while` és `do-while` ciklusok elvileg egymásra visszavezethetők, és bármelyik átírható a másik kettő valamelyikével, ez nem jelenti azt, hogy minden helyzetben mindegy, melyiket használjuk. A legtöbb feladat esetében létezik olyan ciklusszerkezet, amely a problémát a legtermészetesebb módon írja le. A megfelelő ciklus kiválasztása nem a program működését, hanem annak érthetőségét és olvashatóságát befolyásolja.

7.1.1. A `while` ciklus



A `while` ciklust szokás elöl tesztelős ciklusnak is nevezni, hiszen a ciklus feltételét a program minden egyes ismétlés előtt kiértékeli. Amennyiben a feltétel már az első ellenőrzéskor hamis, a ciklustörzs egyszer sem hajtódik végre. Sajátossága, hogy az ismétlések száma általában nem ismert előre, a ciklus futása egy adott állapot fennállásától függ. A `while` ciklust olyan helyzetekben használjuk, amikor egy műveletsort addig kell végrehajtani, amíg egy feltétel igaz, például egy folyamat befejeződéséig, egy adatsor elfogyásáig, vagy addig, amíg a feldolgozás feltétele teljesül.

Időnként szükség lehet egy természetes szám számjegyekre bontására. Ilyen eset például annak vizsgálata, hogy egy szám Harshad-számnak tekinthető-e. Harshad-számnak – más néven Niven-számnak – nevezzük azokat a számokat, amelyek az adott számrendszerben oszthatók saját számjegyeik összegével. A Harshad-szám fogalmát D. R. Kaprekar vezette be (lásd a képet). A „harshad” elnevezés a szanszkrit harṣa szóból származik, amelynek jelentése „nagy öröm” vagy „nagy vidámság”. A Niven-szám elnevezés Ivan Morton Niven nevéhez kapcsolódik, aki e témával foglalkozó munkáját egy 1997-es számelméleti konferencián mutatta be.

Tízes számrendszerben 1 és 10 között minden szám Harshad-szám, a 10-nél nagyobbak közül pedig például a 12, 18, 20, 21, 24, 27, 30, 36, 40, 42, stb. is ebbe a csoportba tartozik.

▦ **Algoritmikai mintázat.** Egy természetes szám *számjegyenkénti feldolgozásának* alapötlete az, hogy a számot ismételten felbontjuk. Az aktuális utolsó számjegyet a maradékos osztás segítségével nyerjük ki (`szam % 10`), majd a számot egész osztással lerövidítjük, eltávolítva a már feldolgozott számjegyet (`szam /= 10`). A folyamatot addig ismételjük, amíg a szám értéke nullára nem csökken (`while (szam > 0)`). Ennek eredményeként a számjegyeket jobbról balra, egyenként dolgozzuk fel.

Az alábbi C kód ellenőrzi, hogy egy szám Harshad-szám-e. A kód magját a szám számjegyösszegét kiszámító `while` ciklus alkotja.

```
#include <stdio.h>

int main() {
    int szam;

    /* A vizsgált szám beolvasása */
    printf("Adj meg egy pozitiv termeszetes szamot: ");
    scanf("%i", &szam);

    int eredeti = szam;   // Az eredeti szám megőrzése
    int osszeg = 0;       // A számjegyek összegének tárolása

    /* A szám számjegyösszegének kiszámítása */
    while (szam > 0) {
        osszeg += szam % 10;  // utolsó számjegy hozzáadása
        szam /= 10;           // a szám lerövidítése
    }

    /* Harshad-tulajdonság ellenőrzése */
    if (eredeti % osszeg == 0) {
        printf("%i Harshad-szam.\n", eredeti);
    } else {
        printf("%i nem Harshad-szam.\n", eredeti);
    }

    return 0;
}
```

Mivel egy szám számjegyenkénti feldolgozásakor addig ismételjük a lépéseket, amíg a szám „elfogy” (értéke nullára csökken), a feladat leprogramozására a while ciklus különösen jól alkalmas.

A témakör begyakorlásához érdemes megoldani az alábbi feladatokat:

- Határozzuk meg egy természetes szám számjegyeinek számát, számjegyeinek átlagát, számjegyeinek szorzatát, prímszámjegyeinek számát, valamint a szám legkisebb és legnagyobb számjegyét.
- Ellenőrizzük, hogy egy szám Armstrong- (más néven narcisztikus) szám-e. Armstrong-számnak nevezünk egy *n* jegyű számot, ha számjegyeinek *n*-edik hatványösszege éppen a számot adja vissza. Például 153 esetén: $153 = 1^3 + 5^3 + 3^3$.
- A tízes számrendszerben összesen 89 Armstrong-szám ismert, ezek közül a legnagyobb 39 jegyű.

Természetesen a `while` ciklust nem csak a számjegyrebontás kontextusában használjuk. A későbbiekben számos további példán keresztül is találkozunk vele.

7.1.2. A `do-while` ciklus

A `do-while` ciklusra gyakran hátul tesztelős cikluskénti utalunk, mivel a ciklus feltételének kiértékelésére a ciklustörzs végrehajtása után kerül sor. Ennek következtében a ciklustörzs legalább egyszer minden esetben végrehajtódik, függetlenül attól, hogy a feltétel teljesül-e. Sajátossága, hogy az ismétlések száma itt sem ismert előre, azonban a döntés csak az első végrehajtás után hozható meg. A `do-while` ciklust olyan helyzetekben használjuk, amikor egy műveletsort legalább egyszer mindenképpen végre kell hajtani, majd az eredmény alapján döntjük el, szükség van-e az ismétlésre.

Az előző fejezetben találkoztunk egy olyan helyzettel, ahol a felhasználónak egy számsorozat bal (1) vagy jobb (2) végéről kellett választania. Az alábbi C kód ezt a lépést robusztusabbá teszi azzal, hogy addig ismétli a választás bekérését, amíg a felhasználó nem 1-et vagy 2-t ad meg:

```
#include <stdio.h>

int main() {
    int valasztas;

    /* A felhasznalo valasztasanak bekérése */
    do {
        printf("Valassz! bal (1) vagy jobb (2): ");
        scanf("%i", &valasztas);
    } while (!(valasztas == 1 || valasztas == 2));
```

```
    printf("A felhasznalo valasztasa: %i\n", valasztas);

    return 0;
}
```

A példában a `do-while` ciklus biztosítja, hogy a kérdés kiírása és a beolvasás legalább egyszer végrehajtódjon. A ciklus feltétele nem a beolvasás sikerességét vizsgálja – ezt megelőlegezzük –, hanem azt, hogy a megadott érték megfelel-e a feladat által előírt tartalmi feltételeknek. Amennyiben a felhasználó nem 1-et vagy 2-t ad meg, a ciklus újra lefut, és a választás bekérése megismétlődik.

A későbbiekben ezzel a ciklusszerkezettel is találkozunk egyéb szövegkörnyezetekben is.

7.1.3. A `for` ciklus

Egy részfeladat megoldása során, amely ismétlést – más szóval iterációt – igényel, gyakran szükség van egy kezdeti előkészületre, valamint az egyes ismétlések közötti állapotváltozás egyértelmű leírására. Ezeket az elemeket a `for` ciklusutasítás egy helyen, áttekinthető formában tartalmazza.

Míg a `while` és a `do-while` ciklusok elsősorban egy feltétel fennállását vizsgálják, addig a `for` ciklus a ciklus vezérléséhez szükséges minden elemet – az inicializálást, a feltételt és az állapotváltozást – egyetlen szerkezetbe foglalja.

Az előző fejezetben már tárgyaltuk a `for` ciklus klasszikusnak nevezhető változatát, amelyben az inicializálás egy ciklusváltozó definiálásából és kezdőértékének megadásából állt. A feltétel annak vizsgálata volt, hogy a ciklusváltozó elérte-e a végső értéket, míg a léptetés során a ciklusváltozó értékét növeltük vagy csökkentettük. Ez az egyszerűsített `for`-mintázat lényegében azt fejezte ki, hogy a ciklusváltozó minden értékére a kezdőértéktől a végső értékig ismételjük a ciklustörzs utasításait. Ez a használati mód jól illeszkedik azokhoz a feladatokhoz, ahol az ismétlések száma előre ismert, azonban a `for` ciklus ennél általánosabb vezérlésre is alkalmas.

A `for` ciklus azonban nem kizárólag ilyen, számláló jellegű feladatokra használható. Általános alakjában az inicializálás, a feltétel és a léptetés tetszőleges kifejezésekből állhat, sőt szükség esetén egyes elemek akár el is hagyhatók. Ez lehetővé teszi, hogy a ciklus vezérlését ne egy külön ciklusváltozó értékei határozzák meg, hanem egy összetettebb állapotváltozás. Az általános `for` ciklus így nem elsősorban az ismétlések számát írja le, hanem azt, hogy milyen feltétel mellett és hogyan haladunk egyik lépésről a másikra.

Az alábbi példából az is kiderül, hogy a `for` ciklus valójában egy elöl tesztelős ciklus, amely működésében megfeleltethető egy `while` ciklusnak, ugyanakkor magába foglalja a szervesen kapcsolódó előkészületi és állapotváltozási lépéseket.

A `for` ciklus segítségével ezek az elemek egyetlen jól áttekinthető szerkezetben jelennek meg.

Következzen a számjegyösszeg számításának `for` ciklusos megvalósítása, ahol az összegváltozó inicializálása előkészületnek, míg a szám fokozatos lerövidítése állapotváltozásnak tekinthető. A lényegi feldolgozás – az aktuális utolsó számjegy kezelése – letisztultan, minden járulékos vezérlési elem nélkül jelenik meg a ciklustörzsben.

```
for (osszeg = 0; szam > 0; szam /= 10) {
    osszeg += szam % 10;
}
```

A következő fejezetben, könyvünk első mesterfejezetében, további lehetőségek tárulnak fel a `for` ciklus használatában, amely a C nyelv kifejező erejének egyik legszebb példája.

7.2. Ugró utasítások ciklusokon belül

A ciklusok működését alapvetően a ciklus feltétele határozza meg: amíg a feltétel igaz, a ciklustörzs újra és újra végrehajtódik. Előfordulhat azonban, hogy a feldolgozás során olyan állapothoz jutunk, amikor a ciklus további futtatása feleslegessé válik, vagy éppen csak az aktuális iteráció végrehajtását szeretnénk kihagyni. Ilyen helyzetekben a ciklus vezérlését futás közben kell módosítanunk.

A C nyelv erre a célra úgynevezett ugró utasításokat biztosít. Ezek lehetővé teszik, hogy a ciklus normál lefutási rendjét megszakítsuk vagy módosítsuk. Ebben az alfejezetben a ciklusokon belül leggyakrabban használt két ugró utasítással, a `break` és a `continue` utasítással ismerkedünk meg. Amint látni fogjuk, tipikus felhasználási helyzetek például: keresés az első találatig → `break;` hibás vagy kihagyandó adat átugrása → `continue`.

7.2.1. A `break` utasítás

A `break` utasítás hatására a program azonnal kilép az aktuális ciklusból, és a vezérlés a ciklust követő utasítással folytatódik. Használatára akkor van szükség, amikor a ciklus célját már elértük, és a további ismétlések végrehajtása nem indokolt.

Tipikusan ilyen helyzet a keresési feladatok esete, amikor a feldolgozás célja egy adott elem első előfordulásának megtalálása. Amint a keresett értéket megtaláltuk, a további vizsgálat feleslegessé válik, ezért a ciklus azonnal megszakítható.

A lineáris keresés algoritmusával korábban, a 6.4. alfejezetben már megismerkedtünk. Ott láttuk, hogy a keresés lényege az elemek sorozatos vizsgálata

mindaddig, amíg a keresett értéket meg nem találjuk, vagy a sorozat végére nem érünk. A `break` utasítás lehetővé teszi, hogy ezt a gondolatot közvetlenül és áttekinthető módon fejezzük ki.

Az alábbi kódban a `tomb[0..n-1]` tárolt számsorozatban keressük a keresett változóban tárolt szám első előfordulásának pozícióját:

```
int pozicio = -1;   // -1 jelzi, hogy nincs találat

for (int i = 0; i < n; i++) {
    if (tomb[i] == keresett) {
        pozicio = i;
        break;      // az első előfordulást megtaláltuk
    }
}
```

A példában a `for` ciklus normál esetben a teljes tömböt bejárná. A break utasítás hatására azonban a ciklus azonnal megszakad, amint a keresett érték első előfordulását megtaláljuk. Ez nemcsak hatékonyabb végrehajtást eredményez, hanem világosan kifejezi az algoritmus célját is. Ez a megoldás egy alternatívája annak a megközelítésnek, amelyet a 6.4. alfejezetben láttunk, ahol a ciklus feltételének módosításával állítottuk le az ismétlést az első találatkor.

7.2.2. A `continue` utasítás

A `continue` utasítás nem szakítja meg a ciklust, csupán az aktuális iteráció végrehajtását hagyja ki. Hatására a ciklus azonnal a következő ismétléssel folytatódik, vagyis a ciklustörzs hátralévő utasításai nem hajtódnak végre az adott lépésben.

Az alábbi példában a szam változó által tárolt természetes szám nem nulla számjegyeinek szorzatát határozzuk meg. Ahelyett, hogy a nem nulla számjegyekre szűrnénk egy `if` feltétellel, a nulla számjegyeket egyszerűen átugorjuk. Bizonyos esetekben ez az – első pillantásra inverznek tűnő – gondolkodásmód átláthatóbb megoldást eredményez.

```
int szorzat = 1;

while (szam > 0) {
    int szamjegy = szam % 10;
    szam /= 10;

    if (szamjegy == 0) {
        continue;   // a nulla számjegyet kihagyjuk
```

```
        }

        szorzat *= szamjegy;
    }
```

A fenti kódban, amikor a ciklus egy nulla számjegyhez ér, az aktuális iteráció további feldolgozása elmarad, és a ciklus a következő számjeggyel folytatódik.

▶ Összegzés

Mivel a megfelelő ciklusszerkezet kiválasztása meghatározza az algoritmus érthetőségét, érdemes az idekapcsolódó utasításokat együtt, egymás mellé állítva áttekinteni, röviden összefoglalva szerepüket:

- `while`: elöl tesztelős ciklus, a ciklustörzs csak akkor hajtódik végre, ha a feltétel igaz
- `do-while`: hátul tesztelős ciklus, ezért a ciklustörzs legalább egyszer lefut
- `for`: általános alakjában az inicializálás, a feltétel és a léptetés egyetlen szerkezetben jelenik meg
- `break`: megszakítja a ciklust (pl. keresés az első találatig)
- `continue`: az aktuális iteráció kihagyását teszi lehetővé

A témakörre további felülnézeti rálátást adnak az alábbi ekvivalenciák, amelyek megmutatják, hogyan valósíthatók meg az egyes ciklusszerkezetek egymás segítségével: a `do-while` szerkezet `while` ciklussal, majd fordítva, valamint a `for` szerkezet `while` ciklussal, majd fordítva. Érdemes lehet elkészíteni ezt a táblázatot logikai sémákkal is.

Eredeti ciklus szintaxisa	Átírva

`do {`
`  /* ciklustörzs */`
`} while (feltétel);`
	`/* ciklustörzs */`
	`while (feltétel) {`
	`  /* ciklustörzs */`
	`}`

`while (feltétel) {`
`  /* ciklustörzs */`
`}`
	`if (feltétel) {`
	`  do {`
	`  /* ciklustörzs */`
	`  } while (feltétel);`
	`}`

`for (kezdőérték-adás; feltétel; lépés) {`
`  /* ciklustörzs */`
`}`
	`kezdőérték-adás;`
	`while (feltétel) {`
	`  /* ciklustörzs */`
	`  lépés;`
	`}`

`while (feltétel) {`
`  /* ciklustörzs */`
`}`
	`for (; feltétel; ) {`
	`  /* ciklustörzs */`
	`}`



AI-csepp – tipikus hibák
Kérd meg az AI-t, hogy szándékosan vétsen tipikus hibákat a fentebb tárgyalt feladatok valamelyikéhez rokon feladat megoldási kódjában. Ezután javítsd ki a hibás kódot, majd kérd meg az AI-t, hogy értékelje a javításodat, és magyarázza el, miért vezetett a hiba helytelen működéshez.

☞ **Tanulság:** a legtöbb ciklussal kapcsolatos hiba nem szintaktikai, hanem logikai természetű; a program helyesen fordul, mégsem azt teszi, amit az algoritmus megkíván.

7.3. BBC szál: legnagyobb közös osztó kiszámítása



A *The Secret Rules of Modern Living: Algorithms* című dokumentumfilmben a narrátor egy téglalap alakú padló lecsempézésének problémáján keresztül mutatja be az euklidészi algoritmust, amely két szám legnagyobb közös osztójának meghatározására szolgál.

A feladat abban áll, hogy a padló ismert méretei alapján meghatározzuk annak a legnagyobb négyzet alakú csempének az oldalhosszát, amellyel a terület hézagmentesen lefedhető. A keresett érték éppen a két oldalhossz legnagyobb közös osztója, amely 150 és 345 esetén 15.

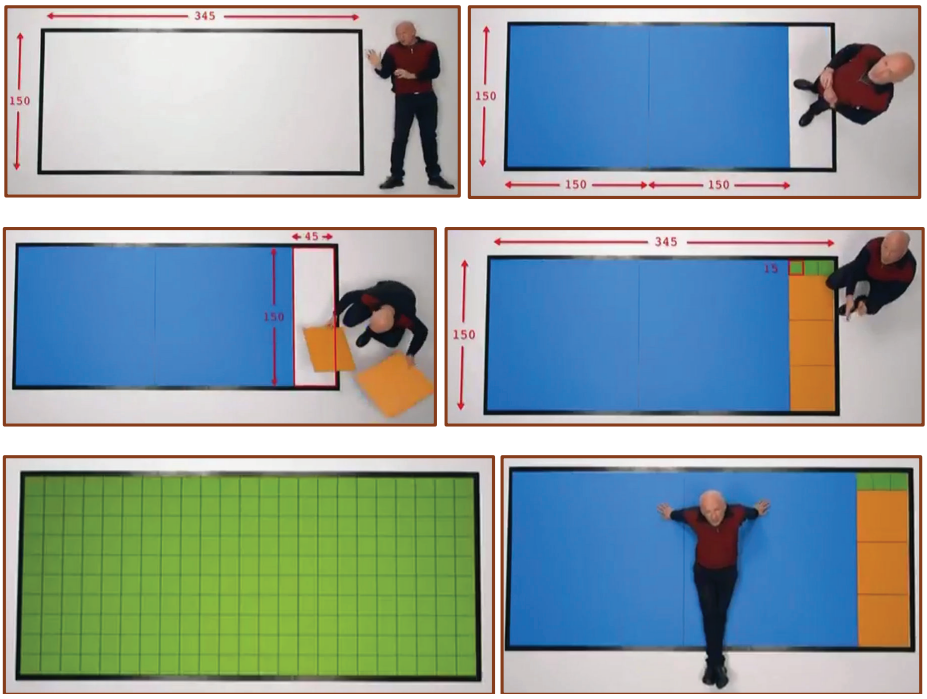
A 345 és 150 oldalhosszak esetén az algoritmus lépései: a 345×150 téglalap lefedésének problémáját visszavezetjük a 150×45 téglalap esetére, ahol 45 = 345 *mod* 150; majd a 150×45 téglalapot redukáljuk a 45×15 téglalapra, ahol 15 = 150 *mod* 45. Mivel a 45 maradék nélkül osztható 15-tel, az algoritmus ezt a lépést követően megáll, és az utolsó nem nulla osztó, a 15 adja az ideális (négyzet alakú) csempe oldalhosszát (7.1. ábra).

Az alábbi kód tetszőleges méretű téglalapra ($a \times b$) határozza meg az ideális csempe oldalhosszát. A továbbiakban feltesszük, hogy $a > b$.

```
int a = 345;
int b = 150;
int r;

/* Feltesszük, hogy a > b > 0 */
while (b != 0) {
    r = a % b;
    a = b;
    b = r;
}
```

```
/* Ekkor a tartalmazza az ideális csempe oldalhosszát */
printf("Az idealis csempe oldalhossza: %i\n", a);
```



7.1. ábra. *Pillanatképek a BBC dokumentumfilmben bemutatott euklidészi algoritmust illusztráló szemléltetésből*

Egy másik módszer a legnagyobb közös osztó meghatározására az úgynevezett kivonásos módszer: addig vonjuk ki a nagyobbik számból a kisebbiket, amíg a két szám egyenlő nem lesz. Ez a végső érték adja a keresett legnagyobb közös osztót. Az erre vonatkozó kód megírását az olvasóra hagyjuk.

A legnagyobb közös osztó meghatározásának „testvérproblémája” a legkisebb közös többszörös kiszámítása: ez felfogható úgy, mint annak a legkisebb magasságnak a meghatározása, amely egész számú *a* és egész számú *b* méretű „téglából” is pontosan felépíthető. Ennek az algoritmusnak az implementálása is maradjon az olvasóra.

Az alapötlet elmélyítése érdekébe próbáld megoldani az alábbi feladatokat is:

- Ha a feladatban az egyetlen megkötés az, hogy a téglalapot négyzet alakú csempékkel fedjük le, amelyek nem feltétlenül azonos méretűek, akkor az euklidészi módszer szerint mindig a lehető legnagyobb négyzetet helyezzük el, majd az eljárást a megmaradó téglalapra ismételjük. Hány csempét

eredményez ez a „mohó módszer”? A 7.1. ábrán bemutatott példa esetében összesen 8 csempét kapunk: 2 kék, 3 narancssárga és 3 zöld négyzetet. Írj programot tetszőleges méretű téglalap esetére, amely meghatározza az euklidészi módszerrel kapott csempék számát!

- Két jelzőfény az indulás pillanatában egyszerre villan fel. Az egyik `x` másodpercenként, a másik `y` másodpercenként villan. Mikor villannak fel újra egyszerre?



AI-csepp

Kérdezd meg az AI-t:

- Az euklidészi módszerre épülő mohó négyzetre bontási algoritmus mindig minimális számú négyzetet eredményez?
- Milyen egyéb módszerek léteznek a legkisebb közös többszörös meghatározására?

7.4. AlgoRythmics szál: bináris keresés



A bináris keresést illusztráló tánc szintén a flamenco stílusából merít. A kontextus megegyezik az előzőekkel: a párját kereső fiú történetét követjük nyomon. Itt is megmarad a „vak számítógép” modellje: mivel minden szereplő a hátán viseli az általa képviselt számot, a fiú – és vele együtt a néző, akinek a fiúval kell azonosulnia – számára rejtve marad a lányok által viselt számsorozat. Ez abból adódik, hogy a lányok a fiúval és a nézővel szemben helyezkednek el (7.2. ábra).

Ezúttal azonban ismert egy fontos tulajdonság a rejtett számsorozatról: az elemek növekvő sorrendben követik egymást. Ez összhangban van a bináris keresés alapfeltételével, miszerint a keresés rendezett számsorozatban történik.

A vak számítógép modellhez igazodva a fiú itt is kénytelen egyenként felkérni a lányokat táncolni, ám most nem csupán azt kell megállapítania, hogy azonos számot viselnek-e, hanem azt is, hogy a lány száma kisebb, nagyobb vagy megegyezik az övével. Ez a különbség a táncban úgy jelenik meg, hogy minden lány ugyanazt a koreográfiát táncolja, mint a fiú, de eltérő sebességgel. A tánc tempója arányos az általuk viselt szám nagyságával.

A fiú valódi párja az a lány, aki pontosan ugyanabban az ütemben táncol, mint ő.

▦ **Algoritmikai mintázat.** A fiú bináris keresési stratégiát alkalmaz. Először a középen álló lányt kéri fel táncolni, majd a közös tánc alapján megállapítja, hogy a lány száma kisebb, nagyobb vagy megegyezik az övével. Egyenlőség esetén a keresés sikeres, és a fiú megtalálta a párját. Ha a fiú száma kisebb a lányénál, akkor

tudja, hogy a bal oldali szakaszon kell folytatnia a keresést, ellenkező esetben pedig a jobb oldali szakaszon. Ezt az eljárást mindaddig ismétli, amíg a párját meg nem találja (az aktuális szakasz középső elemeként), vagy amíg a keresési tartomány ki nem ürül, ami a keresés sikertelenségét jelenti.



7.2. ábra. *Pillanatképek a bináris keresési algoritmust illusztráló flamenco táncból*

Az alábbi implementáció az `a[0..n-1]` növekvő sorrendbe rendezett tömbben keresi binárisan az `x` értéket. Ha a keresett érték nem található meg a számsorozatban, akkor a `pozicio` változó értéke `-1` marad.

```
int bal = 0;
int jobb = n - 1;
int kozep;
int pozicio = -1;

while (bal <= jobb) {
    kozep = bal + (jobb - bal) / 2;

    if (a[kozep] == x) {
        pozicio = kozep;    // x pozíciója
        break;
    } else if (x < a[kozep]) {
```

```
            jobb = kozep - 1;
        } else {
            bal = kozep + 1;
        }
    }
```

7.4.1. Bináris versus lineáris keresés

A bináris keresés hatékonyabb volta a lineáris kereséshez képest abból is nyilvánvaló, hogy egyetlen összehasonlítás után – sikertelenség esetén – a számsorozat felét ki tudjuk zárni, míg a lineáris keresésnél csupán a vizsgált elem kerül kizárásra. Természetesen mindennek megvan az ára: a bináris keresés előfeltétele, hogy a számsorozat rendezett legyen.

Időbonyolultság. Nem nehéz belátni, hogy legjobb esetben a bináris keresés már egyetlen összehasonlítással megtalálja a keresett elemet: ez akkor történik, ha a középső elem éppen a keresett érték. Legrosszabb esetben – amikor a keresett szám nem szerepel a sorozatban – a bináris keresés legfeljebb $log_2 n$ összehasonlítást igényel, ami lényegesen kevesebb, mint a lineáris keresés n összehasonlítása ($O(\log n)$ időbonyolultság $O(n)$-el szemben).

Mikor érdemes bináris keresést alkalmazni? Akkor, ha a számsorozat már eleve rendezett, vagy ha ugyanabban a sorozatban nagyon sokszor kell keresést végezni. Ilyenkor megéri egyszer rendezni a sorozatot, és utána a kereséseket hatékonyan elvégezni.

Ezzel szemben rendezetlen sorozatban végzett egyszeri kereséshez a lineáris keresés a legcélszerűbb, mert a bináris kereséshez szükséges rendezés általában jóval költségesebb művelet, mint maga az egyszeri lineáris keresés.

Egy további érdekes megfigyelés, hogy a bináris keresés esetében a középső elem kiválasztása optimális döntésnek tekinthető a legrosszabb eset szempontjából. Az algoritmus elvileg akkor is működne, ha nem a középső elemet vizsgálnánk meg először, hanem a sorozat bármely más elemét, majd az összehasonlítás eredményétől függően a megfelelő (bal vagy jobb) részszakaszon folytatnánk a keresést.

A középső elem választása azonban biztosítja, hogy a keresési tartomány kettévágásakor a lehető legkisebb legyen az a részszakasz, amely a legrosszabb esetben megmarad. Másképpen fogalmazva: ha egy szakaszt két részre kell bontani, akkor a legnagyobb rész mérete akkor minimális, ha a felosztás a középpontban történik. Ez teszi a bináris keresést optimálissá a legrosszabb esetben.



AI-csepp

Kérdezd meg az AI-t!

- Miért célszerűbb a `kozep = bal + (jobb - bal) / 2` képlettel meghatározni egy szakasz közepét, mint az elsőre természetesebbnek tűnő `kozep = (bal + jobb) / 2` formulával?

- Hogyan kellene módosítani a fenti algoritmust ahhoz, hogy sikertelen keresés esetén ne csupán a sikertelenséget jelezze, hanem a keresett értéknél közvetlenül kisebb, illetve nagyobb elem pozícióját adja meg a számsorozatban?
- Monoton növekvő sorozat esetén hogyan lehetne a bináris keresést úgy módosítani, hogy találat esetén az elem első, illetve utolsó előfordulásának helyét határozza meg?
- Mi az indoklása annak, hogy a bináris keresés időbonyolultsága logaritmikus, azaz $O(\log n)$?

7.5. Hiúz szál: prím-teszt



Monk, a hiúz, egy L ($L \geq 2$) darab deszkából álló egyenes kerítést épít. A kerítés két végét egy-egy oszlop rögzíti. A stabilitás érdekében Monk köztes oszlopokat is kénytelen leásni (legalább egyet), amelyek a kerítést szakaszokra bontják. Monk a szimmetriák megszállottja, ezért a kerítést egyenlő hosszúságú, egész számú deszkából álló szakaszokra szeretné osztani. Ráadásul Monk lusta, ezért a lehető legkevesebb köztes oszlopot szeretné használni.

Például, ha L osztható 2-vel, akkor egyetlen oszlopot ás le középre, így a kerítés két azonos hosszúságú szakaszra bomlik.

Írj C programot, amely beolvassa L értékét, és kiírja a minimálisan szükséges köztes oszlopok számát.

Nem nehéz átlátni, hogy ha a kerítést d egyenlő szakaszra bontjuk, akkor $d-1$ köztes oszlopra van szükség (a két végoszlop adott). A szakaszok száma d csak akkor lehetséges, ha d osztója L-nek, hiszen ekkor minden szakasz hossza L/d (egész) deszka. A minimális szakaszszám (d_{min}) L legkisebb 1-nél nagyobb osztója lesz. Ha L prím, akkor ez az érték éppen L, ami 1 deszkás szakaszokra bontást jelent. A minimálisan szükséges köztes oszlopok száma tehát: $d_{min} - 1$.

Ez a feladat, a legkisebb osztó keresésének problémája, kéz a kézben jár a prímteszttel.

▦ **Algoritmikai mintázat.** Egy $L>1$ szám akkor prím, ha nincs osztója a $2 \leq d \leq \lfloor \sqrt{L} \rfloor$ tartományban. Mivel az osztók párokban jelentkeznek, ezért elegendő 2-től a szám négyzetgyökéig vizsgálni a lehetséges osztókat. A keresés leállítható az első találatnál (`for + break`).

Az alábbi C kód azt ellenőrzi, hogy $L>1$ prím-e.

```
int prim_e = 1;  // megelőlegezzük, hogy prím

for (int i = 2; i * i <= L; i++) {
```

```
        if (L % i == 0) {
            prim_e = 0;  // ha találunk osztót, átállítjuk prim_e-t
            break;
        }
    }

    if (prim_e) {
        printf("%i prim szám\n", L);
    } else {
        printf("%i NEM prim szam\n", L);
    }
```

Egészítsd ki az alábbi kódrészletet a szürke mezők helyén oly módon, hogy az megoldja Monk problémáját, azaz a minimálisan szükséges köztes oszlopszám helyesen kerüljön kiírásra.

```
int d_min = ___;

for (int i = 2; i * i <= L; i++) {
    if (L % i == 0) {
        d_min = ___;
        break;
    }
}

printf("%i koztes oszlop szukseges\n", d_min-1);
```



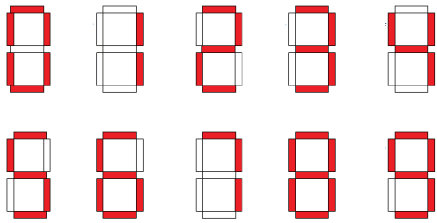
AI-csepp

Kérdezd meg az AI-t, hogy miként tudná Monk tovább gyorsítani a kódját.

7.6. ECN szál: digitális óra



A Sapientia EMTE ECN versenyének 2024-es kiadásán került kitűzésre a Digitális óra (*Clock*) elnevezésű feladat. Adva volt egy digitális óra egy tetszőleges időponttal. Az óra 4 darab, 7 szegmenses LED-kijelzőt használ, és 24 órás formátumban mutatja az időt (0:00–23:59).



7.3. ábra. *A számjegyek megjelenítése egy 7 szegmenses LED-kijelzőn*

A feladat: megkeresni a mai napon azt az időpontot, amely időben a legközelebb van az adotthoz, és amelynél pontosan ugyanannyi LED-szegmens világít, mint az eredeti idő kijelzésénél. Kimenet: H:MM formátumban (óra:perc), ahol az óra nem feltétlenül kétjegyű (0–23), a perc mindig kétjegyű (00–59). Ha nincs olyan (mai napon értelmezett) időpont, ami azonos számú világító szegmenst ad, akkor: Impossible.

Az alábbi kódrészlet a 18:58 időponthoz képest előrefelé haladva megkeresi és kiírja a legközelebbi olyan időpontot, amelynél ugyanannyi LED-szegmens világít; ez 19:08. Mindkét időpontnál 21 szegmens világít. A led tömb az egyes számjegyek 7-szegmenses megjelenítéséhez tartozó világító szegmensek számát tárolja. Itt találkozunk először azzal, hogy egy tömbváltozó már a deklarációkor kezdőértékekkel inicializálható (lásd később). A kód elemzése az olvasóra marad.

```
int led[10] = {6,2,5,5,4,5,6,3,7,6};

int ora = 18, perc = 58;
int inputLedSzam = 0, kurrensLedSzam;

/* bemeneti idő LED-száma */
if (ora < 10)
    inputLedSzam += led[ora];
else
    inputLedSzam += led[ora / 10] + led[ora % 10];

inputLedSzam += led[perc / 10] + led[perc % 10];

/* előrefelé keresés */
do {
    ++perc;
    if (perc == 60) {
        perc = 0;
        ++ora;
    }
```

```
        if (ora == 24)
            break;

        kurrensLedSzam = 0;

        if (ora < 10)
            kurrensLedSzam += led[ora];
        else
            kurrensLedSzam += led[ora / 10] + led[ora % 10];

        kurrensLedSzam += led[perc / 10] + led[perc % 10];

    } while (kurrensLedSzam != inputLedSzam);

    /* kiírás */
    if (ora == 24) {
        printf("Impossible\n");
    } else {
        printf("%i:", ora);
        if (perc < 10) {
            printf("0%i\n", perc);
        } else {
            printf("%i\n", perc);
        }
    }
```



AI-csepp

A Sapientia–ECN versenyen, ha valamely csapat helyes megoldást küld be, akkor az online visszajelzés mellett egy, a feladat színével megegyező színű lufit is kitűznek a csapat asztalára. Ez segít abban, hogy a versenytérben mindenki lássa, mely csapatoknak hány és milyen feladata van megoldva.

Kérdezd meg az AI-t: Miért nem kapna lufit a Clock feladatra az a csapat, amely a fenti megoldást küldi be? Milyen eseteket nem kezel a kód?

►► Zárógondolat

Ebben a fejezetben a ciklusutasításokat nem önálló nyelvi elemekként, hanem algoritmikus eszközökként vizsgáltuk. Láttuk, hogy a `while`, a `do-while` és a `for` ciklusok közötti választás nem a program helyességét, hanem annak érthetőségét, átláthatóságát és gondolati tisztaságát befolyásolja. Ugyanaz az algoritmus többféle ciklusszerkezettel is megvalósítható, de nem mindegy, melyik írja le legtermészetesebben a problémát.

A példákon keresztül megfigyelhettük, hogy a ciklusok gyakran állapotok változását követik nyomon: egy szám „elfogy”, egy keresési tartomány beszűkül,

egy feltétel teljesül vagy megszűnik. A `break` és `continue` utasítások pedig lehetőséget adnak arra, hogy a futás közben reagáljunk ezekre az állapotváltozásokra, és az algoritmus menetét tudatosan módosítsuk.

A fejezet második felében bemutatott algoritmusok – az euklidészi módszer, a bináris keresés, a prímteszt vagy a digitális óra problémája – már azt is megmutatták, hogy a ciklusok a hatékony problémamegoldás alapvető építőkövei. Ezekben a feladatokban nem csupán az a kérdés, hogy „hogyan ismétlünk", hanem az is, hogy mit zárunk ki, mikor állunk meg, és milyen gyorsan jutunk eredményre.

A következő fejezet könyvünk első mesterfejezete lesz. Az első fejezetben megfogalmazott Pareto-elvvel összhangban eddig tudatosan csak a legszükségesebb nyelvi eszközökkel dolgoztunk. A mesterfejezet feladata egyrészt ezeknek a hiányzó elemeknek a pótlása és kiegészítése, másrészt annak bemutatása, hogyan mélyül el a megértés, amikor a vizsgált nyelvi elemek összetettebb, egymással szorosabban összefonódó problémákban jelennek meg.

8. Pointerek és egydimenziós dinamikus tömbök

Eddigi programjainkban a változókat főként a `nev`ükön keresztül értük el. Ha például volt egy `int x` változónk, akkor az `x` névvel hivatkoztunk az értékére. A C nyelv azonban azt is lehetővé teszi, hogy ne közvetlenül a változó nevével, hanem a memóriabeli címével dolgozzunk. Ehhez használjuk a pointereket, vagyis mutatókat.

A pointer elsőre szokatlan fogalom, de a mögötte álló gondolat egyszerű: egy pointer olyan változó, amely egy másik változó címét tárolja. Ha ismerjük egy változó címét, akkor ezen a címen keresztül elérhetjük, sőt módosíthatjuk is az ott tárolt értéket. Ebben a fejezetben ezt először egyszerű `int` és `float` változókon figyeljük meg, hogy a cím, a mutató és az indirekt elérés fogalma világos legyen.

Ezután megnézzük, hogyan kapcsolódnak a pointerek az egydimenziós tömbökhöz. Látni fogjuk, hogy egy tömb neve sok helyzetben az első elem címeként viselkedik, és hogy a tömbelemeket pointereken keresztül is elérhetjük. Ez a gondolat készíti elő a dinamikus memóriafoglalást: amikor a program futás közben dönti el, mekkora tömbre van szüksége.



A C szálon először a pointerek alapgondolatával ismerkedünk meg: mit jelent egy változó címe, hogyan tárolhatjuk ezt a címet egy pointerben, és hogyan érhetjük el az értéket közvetetten, a címen keresztül. Ezután megnézzük, hogyan kapcsolódnak a pointerek az egydimenziós tömbökhöz, majd erre építve bevezetjük a dinamikus memóriafoglalást. A fejezet főszereplői a címképző `&` operátor, az indirekciós `*` operátor, a tömbnév és az első elem címe közötti kapcsolat, valamint a `malloc`, a `calloc` és a `free` függvények lesznek.

8.1. Mutatók

8.1.1. Változók címe

Amikor létrehozunk egy változót, a program memóriaterületet foglal le számára. A változó neve segítségével az ott tárolt értékre hivatkozunk.

```
int x = 1972;
```

Ebben a sorban az `x` változó egy egész számot tárol. A változónak azonban nemcsak értéke van, hanem memóriacíme is: valahol a memóriában található az a hely, ahol az 1972 érték tárolódik.

A változó címét az `&` operátorral kérdezhetjük le.

```
printf("%p\n", &x);
```

A `%p` formátumjel memóriacím kiírására szolgál. A kiírt cím konkrét értéke futásonként és gépenként változhat. Számunkra most nem az a fontos, hogy pontosan milyen szám jelenik meg, hanem az, hogy a változónak van címe, amelyre hivatkozni tudunk.

8.1.2. Pointer deklarációja

Egy pointer olyan változó, amely memóriacímet tárol. A pointer deklarációjában a típus után `*` jelet írunk.

```
int *p;
```

Ez azt jelenti, hogy `p` olyan pointer, amely `int` típusú változó címét képes tárolni.

```
int x = 1972;
int *p;

p = &x;
```

A `p = &x;` értékadás után a `p` pointer az `x` változó címét tárolja. Ezt úgy is mondhatjuk, hogy `p` az `x` változóra mutat.

Hasonlóan deklarálhatunk `float` típusú változóra mutató pointert is.

```
float atlag = 8.5;
float *q;

q = &atlag;
```

A pointer típusa azt mondja meg, hogy milyen típusú adat található azon a címen, amelyre a pointer mutat. Az `int *` típusú pointer `int` típusú változóra, a `loat *` típusú pointer `float` típusú változóra mutat.

[i] Megjegyzés: A `*` jel a deklarációban a változóhoz tartozik. Ez különösen akkor fontos, ha több változót deklarálunk egy sorban. Például az `int *p, x;`

deklaráció nyomán `p` pointer, de `x` egyszerű `int` változó. Ha két pointert szeretnénk deklarálni, mindkét név elé ki kell írni a `*` jelet: `int *p1, *p2;`

8.1.3. A `&` és a `*` operátor

A pointerek használatát két fontos operátor segíti: (i) a címképző `&` operátor egy változó címét adja meg; (ii) az indirekciós vagy dereferáló `*` operátor a pointer által mutatott helyen tárolt értéket éri el.

```
int x = 1978, y, *p;

p = &x;
y = *p;
```

Mivel `p` az `x` változó címét tárolja, a `*p` kifejezés az `x` értékét jelenti. A második értékadás után `y` ugyanazt az értéket kapja, mint `x`.

A dereferálás nemcsak olvasásra, hanem írásra is használható:

```
*p = 1989;
```

Ez a sor nem a pointer értékét módosítja, hanem azt az értéket, amelyre a pointer mutat. Mivel `p` az `x` címét tárolja, a fenti sor az `x` változó értékét módosítja.

Íme egy további szemléltető példa:

```
#include <stdio.h>

int main() {
    int x = 1978;
    int *p;

    p = &x;

    printf("%i\n", x);
    printf("%i\n", *p);

    *p = 1989;

    printf("%i\n", x);
    printf("%i\n", *p);

    return 0;
}
```

A programban az `x` és a `*p` ugyanarra a memóriaterületre hivatkozik. Az `x` névvel közvetlenül érjük el az értéket, a `*p` kifejezéssel pedig a címen keresztül.

8.1.4. Több pointer ugyanarra a változóra

Több pointer is mutathat ugyanarra a változóra.

```
int x = 1990;
int *p1, *p2;

p1 = &x;
p2 = &x;
```

A `p1` és a `p2` két külön pointerváltozó, de ugyanazt a címet tárolják: az `x` változó címét.

```
*p1 = 1991;
*p2 = 1992;
```

Mindkét értékadás az `x` változó értékét módosítja, mert mindkét pointer ugyanarra a változóra mutat.

Két pointer össze is hasonlítható.

```
if (p1 == p2) {
    printf("Ugyanarra a valtozora mutatnak.\n");
}
```

Két pointer akkor egyenlő, ha ugyanazt a címet tárolja.

[i] Megjegyzés: A pointerek egyenlőségének vizsgálata általában jól érthető: azt kérdezzük, hogy ugyanarra az objektumra mutatnak-e. A kisebb-nagyobb összehasonlításokkal azonban óvatosan kell bánni. Ezeknek igazán csak ugyanazon tömb elemei között van jól értelmezhető jelentésük. Erre a következő alfejezetben térünk vissza.

8.1.5. A `NULL` pointer

Előfordulhat, hogy egy pointer még nem mutat érvényes változóra. Ilyenkor nem szabad véletlen címet hagyni benne. Jobb, ha egyértelműen jelezzük, hogy a pointer jelenleg nem mutat sehova. Erre szolgál a `NULL`.

```
int *p = NULL;
```

A `NULL` pointer nem mutat érvényes adatra, ezért nem szabad dereferálni.

```
int *p = NULL;

/* Hibás lenne:
   printf("%i\n", *p);
*/
```

▶ **Összegzés.** A pointerek lehetővé teszik, hogy a program ne csak a változók nevén, hanem a memóriacímükön keresztül is elérje az adatokat. A `&` operátor címet képez, a `*` operátor pedig a pointer által mutatott helyen tárolt értéket éri el.

```
&x      az x változó címe
p       egy pointer, amely címet tárol
*p      a p által mutatott helyen lévő érték
```

A biztonságos pointerhasználat kulcsa az, hogy mindig tudjuk, mire mutat a pointer, és csak akkor dereferáljuk, ha valóban érvényes memóriaterületre mutat.



AI-csepp
Kérdezd meg az AI-t!
- Melyek a leggyakoribb kezdő pointerhibák C nyelvben?
- Mi a különbség az x, az &x, a p és a *p között, ha p = &x?
- Mi történhet, ha egy inicializálatlan pointert dereferálunk?

8.2. A pointerek és az egydimenziós tömbök kapcsolata

A C nyelv egyik legfontosabb sajátossága, hogy a tömbök és a pointerek szoros kapcsolatban állnak egymással. Ez a kapcsolat elsőre furcsának tűnhet, de nagyon sok C program működésének alapja.

Fontos azonban már az elején kimondani: a tömb és a pointer nem ugyanaz. Egy tömb valódi adatterület, amely több elemet tartalmaz. Egy pointer ezzel szemben egy címet tároló változó. A kettő mégis sok helyzetben hasonlóan használható.

8.2.1. A tömb neve és az első elem címe

Vegyük a következő tömböt:

```
int a[5] = {10, 20, 30, 40, 50};
```

A tömb elemeit eddig indexeléssel értük el, de a C nyelvben egy tömb neve a legtöbb kifejezésben a tömb első elemének címeként viselkedik. Ezért írhatjuk például:

```
int *p;

p = a;
```

Ez azt jelenti, hogy `p` az a tömb első elemére mutat. A fenti értékadás ugyanazt a címet adja a pointernek, mint amit az `&a[0]` kifejezéssel kapnánk.

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

A két értékadás hatása ebben az esetben ugyanaz.

8.2.2. Indexelés pointeren keresztül

Ha `p` az `a` tömb első elemére mutat, akkor a tömb elemeit a pointeren keresztül is elérhetjük.

```
int a[5] = {10, 20, 30, 40, 50};
int *p;

p = a;

printf("%i\n", p[0]);
printf("%i\n", p[3]);
```

A `p[0]` ugyanazt az elemet jelenti, mint az `a[0]`, a `p[3]` pedig ugyanazt, mint az a[3]. Ez azért lehetséges, mert az indexelés valójában pointeraritmetikára épül (lásd lentebb).

```
a[i]  ≡  *(a + i)
p[i]  ≡  *(p + i)
```

Vagyis az `a[i]` kifejezés jelentése: induljunk el az `a` által jelölt címtől `i` elemmel előrébb, majd érjük el az ott található értéket.

8.2.3. Pointeraritmetika tömbökben

A pointeraritmetika csak akkor érthető biztonságosan, ha összefüggő memóriaterületen, például tömbön belül gondolkodunk.

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a;

printf("%i\n", *p);
printf("%i\n", *(p + 1));
printf("%i\n", *(p + 2));
```

A `p + 1` nem egy bájttal lép tovább, hanem egy `int` méretű elemmel. Ha `p` az `a[0]` elemre mutat, akkor `p + 1` az `a[1]` elem címe, `p + 2` pedig az `a[2]` elem címe.

A lépés mérete mindig a pointer típusától függ.

```
int * esetén      1 lépés = sizeof(int) bájt
float * esetén    1 lépés = sizeof(float) bájt
```

Ezért fontos, hogy a pointer típusa ne csak azt mondja meg, milyen címet tárolunk, hanem azt is, hogyan kell azon a címen az adatot értelmezni, és mekkora lépésekben haladunk a pointeraritmetika során.

A pointer növelhető is.

```
int a[5] = {10, 20, 30, 40, 50};
int *p = a;

p++;
printf("%i\n", *p);
```

A `p++` után a pointer már nem az `a[0]`, hanem az `a[1]` elemre mutat. Ez hasznos lehet bejáráskor, de óvatosan kell használni: ha elveszítjük az eredeti címet, később nem biztos, hogy vissza tudunk térni a tömb elejére.

8.2.4. Miben különbözik a tömbnév és a pointer?

Bár a tömbnév sok helyzetben pointerként viselkedik, nem pointerváltozó.

```
int a[5];
int *p;

p = a;
```

Ez helyes: a `p` pointer megkapja az `a` tömb első elemének címét.
Fordítva azonban nem írhatjuk:

```
/* Hibás:
   a = p;
*/
```

Az `a` nem változóként módosítható pointer. A tömb neve a tömbhöz kötött. Nem lehet átállítani, hogy másik tömbre mutasson.

Hasonlóan, a `p++;` megengedett, mert `p` pointerváltozó, de az `a++;` nem megengedett, mert `a` tömbnév, nem módosítható pointerváltozó.

A különbség a `sizeof` operátorral is látható.

```
int a[5];
int *p = a;

printf("%i\n", sizeof(a));
printf("%i\n", sizeof(p));
```

A `sizeof(a)` a teljes tömb méretét adja meg bájtban, vagyis `5 * sizeof(int)` értéket. A `sizeof(p)` viszont csak a pointerváltozó méretét adja meg. Ez a cím tárolásához szükséges méret, nem a tömb mérete.

[i] Megjegyzés: Egy adott rendszeren belül a pointerek mérete gyakran azonos, függetlenül attól, hogy `int`, `float` vagy más típusú objektumra mutatnak. A `sizeof(*p)` viszont a mutatott objektum méretét adja meg. Ha `p` típusa `int *`, akkor `sizeof(*p)` ugyanaz, mint `sizeof(int)`.

8.2.5. Tömb bejárása pointerrel

Egy tömböt bejárhatunk indexeléssel:

```
int a[5] = {10, 20, 30, 40, 50};

for (int i = 0; i < 5; i++) {
    printf("%i\n", a[i]);
}
```

Ugyanez pointerrel is megtehető.

```
int a[5] = {10, 20, 30, 40, 50};
```

```
for (int *p = a; p - a < 5; p++) {
    printf("%i\n", *p);
}
```

A második változatban a ciklus elején `p` az `a` tömb első elemére mutat. Minden lépésben kiírjuk a `p` által mutatott értéket, majd a `p++` művelettel a következő elemre lépünk. A feltételben szereplő `p - a` azt mutatja meg, hány `int` elemnyi távolságra van a `p` pointer a tömb elejétől. A ciklus addig fut, amíg ez a távolság kisebb mint 5.

▶ **Összegzés.** A C nyelvben a tömbök és a pointerek közötti kapcsolat nem véletlen, hanem a nyelv memóriamodelljéből következik. Egy egydimenziós tömb elemei egymás után helyezkednek el a memóriában, a tömb neve pedig sok kifejezésben az első elem címeként viselkedik.

```
a[i]      ugyanazt jelenti, mint *(a + i)
p[i]      ugyanazt jelenti, mint *(p + i)
a         sok helyzetben az első elem címe
p         pointerváltozó, amely módosítható
```

Ez a kapcsolat készíti elő a következő lépést: a dinamikus memóriafoglalást. Ott olyan egydimenziós tömbökkel fogunk dolgozni, amelyeknek nincs hagyományos tömbnevük. Csak egy pointerünk lesz, amely az első elemük címét tárolja.



AI-csepp

Kérdezd meg az AI-t!

- Az összeadás kommutativitásából adódóan `*(a + i)` ugyanaz, mint `*(i + a)`. Következik-e ebből, hogy az `a[i]` tömbelem-hivatkozás lecserélhető `i[a]` alakra? Indokold a választ!
- Miben hasonlít és miben különbözik egy tömb neve és egy pointerváltozó?
- Miért más a `sizeof(a)` és a `sizeof(p)` értéke, ha `p = a`?

8.3. Dinamikus memóriafoglalás egydimenziós tömböknek

Eddig kétféle tömbbel találkoztunk. Volt olyan tömb, amelynek méretét már a program szövegében megadtuk:

```
int a[100];
```

és volt olyan is, amelynek méretét futás közben olvastuk be:

```
int n;

scanf("%i", &n);

int a[n];
```

Ez utóbbi alak egyszerű programokban kényelmes lehet, de nem minden helyzetben elegendő vagy biztonságos. A dinamikus memóriafoglalás rugalmasabb és robusztusabb megoldást ad: a program futás közben, könyvtári függvény segítségével kérhet memóriaterületet akkora tömb számára, amekkorára ténylegesen szüksége van. Ha a területre már nincs szükség, azt szintén könyvtári függvénnyel szabadíthatjuk fel.

8.3.1. A `malloc` és `calloc` függvények

Dinamikus memóriafoglaláshoz gyakran a `malloc` függvényt használhatjuk. Ehhez be kell kapcsolnunk az `stdlib.h` fejlécállományt: `#include <stdlib.h>`

A `malloc` paramétere az, hogy hány bájt memóriát szeretnénk foglalni. Ha `n` darab `int` elemnek szeretnénk helyet, akkor ennyi bájtra van szükségünk: `n * sizeof(int)`

A foglalás alakja:

```
int *a;
a = malloc(n * sizeof(int));
```

A `malloc` egy memóriacímet ad vissza: annak a lefoglalt területnek a kezdőcímét, ahol az elemek egymás után elférnek. Ezt a címet tároljuk az `a` pointerben.

A dinamikusan foglalt tömbnek nincs olyan neve, mint egy hagyományos tömbnek. Csak az első elem címét tároljuk egy pointerben. Mivel azonban a pointer indexelhető, a lefoglalt terület elemeit ugyanúgy elérhetjük `a[i]` alakban, mintha hagyományos tömbbel dolgoznánk.

```
a[0] = 10;
a[1] = 20;
a[2] = 30;
```

A `calloc` szintén dinamikus memóriafoglalásra szolgál. Egyik fontos tulajdonsága, hogy a lefoglalt memóriaterületet nullázza. Abban is különbözik a `malloc` függvénytől, hogy két paramétere van: hány elemet szeretnénk foglalni, és mekkora egy elem mérete.

```
int *a;
a = calloc(n, sizeof(int));
```

Ez a hívás `n` darab `int` méretű elemnek foglal helyet, és az elemek kezdeti értéke `0` lesz.

[i] Megjegyzés: Gyakori C-s megoldás az is, hogy nem a típust, hanem a pointer által mutatott objektum méretét írjuk ki:

```
a = malloc(n * sizeof(*a));
```

Mivel `a` típusa `int *`, a `*a` típusa `int`, ezért a `sizeof(*a)` ugyanazt a méretet adja, mint a `sizeof(int)`. Ez az alak akkor hasznos, ha később megváltoztatjuk a pointer típusát. Ilyenkor kisebb az esélye annak, hogy a foglalásban elfelejtjük átírni a típust.

8.3.2. Sikertelen memóriafoglalás

A memóriafoglalás nem mindig sikerül. Ha a program túl sok memóriát kér, vagy valamilyen más okból nem kaphatja meg a kért területet, a `malloc` `NULL` értékkel tér vissza.

Ezért a foglalás után ellenőrizni kell a pointert.

```
a = malloc(n * sizeof(int));

if (a == NULL) {
    printf("Sikertelen memoriafoglalas.\n");
    return 1;
}
```

Csak akkor szabad használni az `a[i]` alakot, ha a foglalás sikeres volt.

8.3.3. Dinamikus tömb beolvasása és kiírása

Nézzünk egy teljes példát. A program először beolvassa, hány egész szám következik. Ezután pontosan ekkora dinamikus tömböt foglal, beolvassa az elemeket, majd kiírja őket.

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

int main() {
    int n;
    int *a;
```

```
    scanf("%i", &n);

    a = malloc(n * sizeof(int));

    if (a == NULL) {
        printf("Sikertelen memoriafoglalas.\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        scanf("%i", &a[i]);
    }

    for (int i = 0; i < n; i++) {
        printf("%i ", a[i]);
    }

    printf("\n");

    free(a);

    return 0;
}
```

Figyeljük meg a program fő lépéseit:

```
1. Beolvassuk az elemszámot.
2. Helyet foglalunk n darab int számára.
3. Ellenőrizzük, hogy sikerült-e a foglalás.
4. Beolvassuk az elemeket.
5. Feldolgozzuk vagy kiírjuk az elemeket.
6. Felszabadítjuk a lefoglalt memóriát.
```

Vegyük észre, hogy a dinamikus tömböt ugyanúgy feldolgozhatjuk ciklussal, mint a hagyományos tömböt. Az `a[i]` alak ugyanúgy használható, mintha előre deklarált tömbbel dolgoznánk. A különbség az, hogy a tömb méretét futás közben döntöttük el, a memóriaterületet pedig nekünk kell felszabadítanunk, amikor már nincs rá szükség.

A `free(a);` hívás erősen ajánlott. A dinamikusan lefoglalt memóriát a programozónak kell felszabadítania, amikor már nincs rá szükség.

8.3.4. A `free` függvény

A `free` felszabadítja a korábban lefoglalt memóriaterületet.

```
free(a);
```

A `free` után az `a` pointer értéke ugyan még tartalmazhatja a régi címet, de az a memóriaterület már nem a programé. Ezért a felszabadítás után nem szabad használni az `a[i]` elemeket.

```
free(a);

/* Hibás lenne:
   printf("%i\n", a[0]);
*/
```

Ha szeretnénk jelezni, hogy a pointer már nem mutat érvényes területre, beállíthatjuk `NULL` értékre.

```
free(a);
a = NULL;
```

Ez nem kötelező, de bizonyos helyzetekben segíthet elkerülni a véletlen újrahasználatot.

[i] Megjegyzés: Csak olyan címet szabad átadni a `free` függvénynek, amelyet korábban dinamikus memóriafoglaló függvény, például `malloc` vagy `calloc` adott vissza, és amelyet még nem szabadítottunk fel.

▶ **Összegzés.** A dinamikus memóriafoglalás lehetővé teszi, hogy a program futás közben kérjen memóriaterületet. Ez különösen hasznos akkor, ha a szükséges tömbméret csak a bemenetből derül ki. A dinamikus tömbnek nincs hagyományos tömbneve. Egy pointer tárolja az első elem címét, és ezen keresztül érjük el az elemeket. Ez a gondolat köti össze a fejezet témáit: a pointer címet tárol, a tömb elemei egymás után helyezkednek el, a dinamikusan foglalt memóriaterület pedig pointeren keresztül használható egydimenziós tömbként.



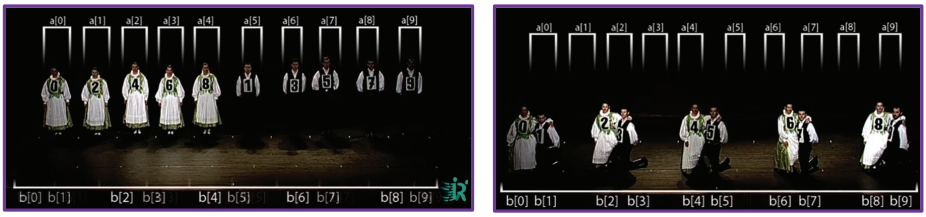
AI-csepp
Kérdezd meg az AI-t!
- Melyek a tipikus hibák dinamikus memóriafoglalásnál?
- Melyik gyorsabb: a `malloc` vagy a `calloc`? Mitől függhet a válasz?
- Mikor célszerű `malloc`-ot, és mikor `calloc`-ot használni?

- Mit jelent a memóriaszivárgás, és hogyan kerülhető el?
- Miért lehet célszerűbb dinamikus memóriafoglalást használni az `int a[n]` alak helyett?

8.4. AlgoRythmics szál: két rendezett sorozat összefésülése



Az AlgoRythmics szálban később találkozni fogunk egy fontos rendezési módszerrel, az összefésüléses rendezéssel. Ennek egyik alaplépése az összefésülés: két már rendezett sorozatból egyetlen rendezett sorozatot készítünk (8.1. ábra).



8.1. ábra. *Szemléltető jelentek az összefésüléses rendezést ábrázoló AlgoRythmics videóból. Az* `a[0..4]` *és* `a[5..9]` *rendezett tömbszakaszok kerülnek összefésülésre a* `b[0..9]` *tömbbe.*

Most még nem a teljes rendezési algoritmust vizsgáljuk, hanem csak ezt az alapműveletet. Ez jól illik a fejezet témájához, mert a két bemeneti sorozatot dinamikus tömbökben tárolhatjuk, az eredményt pedig egy harmadik dinamikus tömbben hozzuk létre. A feldolgozás közben a tömbökön pointerekkel lépegetünk végig.

Tegyük fel, hogy két növekvő sorrendbe rendezett egész számsorozatunk van:

```
2 5 9 14
1 3 7 10 12
```

Az összefésülés eredménye:

```
1 2 3 5 7 9 10 12 14
```

A feladat tehát nem az, hogy a két tömböt külön-külön rendezzük. Ezt most adottnak tekintjük. A cél az, hogy a két már rendezett tömbből egy harmadik, szintén rendezett tömböt állítsunk elő.

▦ **Algoritmikus mintázat.** Az összefésülés során mindig a két bemeneti tömb aktuális elemét hasonlítjuk össze. A kisebbiket bemásoljuk az eredménytömb következő helyére, majd abban a tömbben lépünk tovább, ahonnan az elemet kivettük. Ha az egyik tömb elfogyott, a másik tömb maradék elemeit egyszerűen átmásoljuk.

```
1. Álljunk mindkét bemeneti tömb elejére.
2. Hasonlítsuk össze az aktuális elemeket.
3. A kisebbiket másoljuk az eredménytömb következő helyére.
4. Lépjünk tovább abban a tömbben, ahonnan másoltunk.
5. Ismételjük, amíg valamelyik bemeneti tömb el nem fogy.
6. Másoljuk át a másik tömb maradékát.
```

Az alábbi programrészletben három dinamikus tömbbel dolgozunk. Az `a` és `b` tömb már rendezett, rendre `n` és `m` eleműek, a `c` tömbben pedig létrejön az összefésült eredmény. Feltételezzük, hogy a `c` tömbnek már lefoglaltunk `n + m` elemnyi helyet.

```
int *pa = a;
int *pb = b;
int *pc = c;

while (pa < a + n && pb < b + m) {
    if (*pa < *pb) {
        *pc = *pa;
        pa++;
    } else {
        *pc = *pb;
        pb++;
    }

    pc++;
}

while (pa < a + n) {
    *pc = *pa;
    pa++;
    pc++;
}

while (pb < b + m) {
    *pc = *pb;
    pb++;
    pc++;
}
```

A függvényben a `pa`, `pb` és `pc` pointerek rendre az `a`, `b` és `c` tömb aktuális elemére mutatnak. A `pa++`, `pb++` és `pc++` műveletek nem egyetlen bájttal lépnek tovább, hanem a megfelelő típus, jelen esetben az `int` méretével. Ezért a pointerek mindig a következő tömbelemre mutatnak.

A ciklusfeltételekben az `a + n` az `a` tömb utolsó eleme utáni pozíciót jelöli. Hasonlóan, a `b + m` a `b` tömb utolsó eleme utáni pozíció. Ezekre a címekre már nem szabad dereferálással hivatkozni, vagyis hibás lenne a `*(a + n)` vagy a `*(b + m)` kifejezés. Összehasonlításra azonban használhatók: azt jelzik, hol ér véget a bejárás.

A teljes program megírását az olvasóra hagyjuk. A teendők:

- Deklaráljuk a három `int *` pointert: `a`, `b` és `c`, valamint az `n` és `m` változókat.
- Beolvassuk az `n` és `m` értékét.
- A `malloc` segítségével helyet foglalunk a három dinamikus `int` tömbnek: rendre `n`, `m`, illetve `n + m` elem számára. Mindegyik helyfoglalás után ellenőrizzük, hogy sikeres volt-e.
- Beolvassuk a rendezett számsorokat az `a` és `b` tömbbe.
- A fentebb megírt algoritmus segítségével elvégezzük az összefésülést.
- Kiírjuk a `c` tömb elemeit.
- Három `free` hívással felszabadítjuk a lefoglalt memóriaterületeket.

Az összefésülés időbonyolultsága `O(n + m)`, azaz lineáris, mert minden elemet pontosan egyszer másolunk át az eredménytömbbe. Ezért ez az egyszerű művelet nagyon fontos építőelem: később az összefésüléses rendezésben éppen ezt használjuk majd fel arra, hogy két rendezett részsort egyetlen rendezett sorrá kapcsoljunk össze.



AI-csepp

Kérdezd meg az AI-t!

- Vezesd végig az összefésülés algoritmusát a felvezetőben szereplő példán! Minden lépésben mutasd meg, mire mutat a `pa`, a `pb` és a `pc` pointer.
- Miért elegendő két rendezett tömb összefésüléséhez mindig csak az aktuális elemeket összehasonlítani?
- Hogyan kezeli az algoritmus, ha a rendezett számsorozatokban közös elemek vannak?
- Mi történik akkor, ha a számsorozatok nem szigorúan növekvők, hanem csak monoton növekvők, vagyis ismétlődő elemeket is tartalmazhatnak?
- Mi változik, ha a sorozatok csökkenő sorrendbe rendezettek?

8.5. Hiúz szál: szétválogatás helyben



Egy sportrendezvényre érkező látogatók sorszámot kapnak. A szervezők a beléptetés előtt két csoportra szeretnék bontani a sorszámokat: a páratlan sorszámú látogatók az egyik kapuhoz, a páros sorszámúak a másik kapuhoz kerülnek.

A sorszámok egy `n` elemű, dinamikusan foglalt `t` tömbben vannak. A feladatot memóriatakarékosan szeretnénk megoldani: nem hozunk létre külön tömböt a páratlan és külön tömböt a páros sorszámoknak, hanem az eredeti tömbön belül válogatjuk szét az elemeket.

Például ha a sorszámok kezdetben így állnak:

```
12 5 18 7 3 16 9 4
```

akkor a szétválogatás után kaphatunk például ilyen elrendezést:

```
9 5 3 7 18 16 12 4
```

Ez nem teljes rendezés. Nem az a cél, hogy a sorszámok növekvő sorrendbe kerüljenek. Csak azt szeretnénk elérni, hogy a páratlan számok egy összefüggő szakaszt alkossanak a tömb elején, a páros számok pedig egy másik összefüggő szakaszt a tömb végén.

Feladat: Írjunk programot, amely beolvas egy `n` elemű természetes számsorozatot egy dinamikusan foglalt `t` tömbbe, majd helyben szétválogatja az elemeket páratlan és páros számokra.

A megoldás követelményei:

- a program dinamikusan foglalja le a `t` tömböt;
- ne használjon külön tömböt a páratlan és páros elemeknek;
- a szétválogatást pointerekkel végezze;
- a páratlan számok kerüljenek a tömb elejére;
- a páros számok kerüljenek a tömb végére;
- a program határozza meg, hány páratlan és hány páros szám van.

Ötlet: Használjunk két pointert. Az egyik pointer, legyen `bal`, a tömb elejéről indul. A másik pointer, legyen `jobb`, a tömb végéről indul. A `bal` pointer addig lépjen jobbra, amíg páratlan számot lát, hiszen ezek jó helyen vannak a tömb elején. A `jobb` pointer addig lépjen balra, amíg páros számot lát, hiszen ezek jó helyen vannak a tömb végén. Ha a `bal` pointer páros számnál áll meg, a `jobb` pointer pedig páratlan számnál, akkor a két érték rossz oldalon van. Ilyenkor cseréljük ki őket. A folyamat addig tart, amíg a két pointer találkozik, vagy keresztezi egymást.

A szétválogatás után a két csoport határa egy pointerrel jelölhető. A páratlan szakasz a tömb elején, vagyis a `t` címen kezdődik. A páros szakasz pedig a `hatar` címen kezdődik.

```
[t, hatar)        páratlan számok
[hatar, t + n)    páros számok
```

A páratlan számok száma pointerkülönbséggel számítható ki:

```
int paratlanok_szama = hatar - t;
```

A páros számok száma pedig:

```
int parosok_szama = n - paratlanok_szama;
```

[i] Megjegyzés. A `hatar` pointer lehet maga a `t` is, ha nincs páratlan szám, és lehet `t + n` is, ha nincs páros szám. Ezért a `hatar` nem feltétlenül dereferálható pointer. A szerepe itt az, hogy kijelölje a két szakasz közötti határt.

Ez a feladat jól előkészíti a gyorsrendezés gondolatát is, amelyet majd a 15. fejezetben tárgyalunk részletesen. Ott szintén egy tömböt válogatunk szét két részre egy szabály alapján, csak nem páratlanság és párosság szerint, hanem egy választott vezérelemhez viszonyítva.



AI-csepp

Kérdezd meg az AI-t!

- Vezesd végig az algoritmust a felvezetőben szereplő példán!
- Mennyi az időbonyolultsága a szétválogatás algoritmusnak?
- Miben különbözik az itt tárgyalt szétválogatás a gyorsrendezés keze alá dolgozó szétválohatásnál?

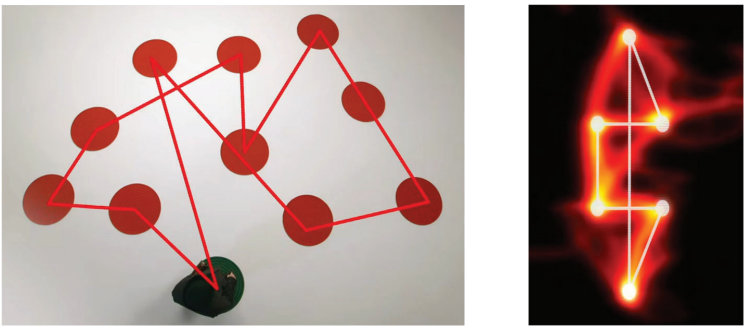
8.6. BBC szál: az utazóügynök és a mohó útvonal



A BBC-filmben az utazóügynök problémája nem csupán matematikai fejtörőként jelenik meg. A kérdésnek gyakorlati jelentősége is van: hogyan lehet több helyszínt úgy bejárni, hogy az út minél rövidebb legyen? Hasonló gondolat jelenik meg csomagküldésnél, szállításnál, útvonaltervezésnél, sőt olyan helyzetekben is, amikor nem valódi városokat, hanem valamilyen feldolgozási pontokat kell optimális sorrendbe rendezni.

A feladat lényege egyszerűen megfogalmazható: adott néhány város, és olyan körutat szeretnénk találni, amely minden várost pontosan egyszer érint, majd visszatér a kiinduló városba (8.2. ábra). A cél az, hogy a körút teljes hossza minél kisebb legyen.

A probléma jelentőségét az is mutatja, hogy szorosan kapcsolódik a számítástudomány egyik legismertebb nagy kérdéséhez: vajon az ilyen nehéz keresési feladatokra létezik-e minden esetben gyors, hatékony módszer? Ennek a kérdéskörnek a megoldására egymillió dolláros díjat is kitűztek. A BBC-filmben az is előkerül, hogy a természetben is találunk érdekes stratégiákat: például a méheknek is hatékony útvonalakat kell kialakítaniuk a virágok között (8.2. ábra).



(a) (b)

8.2. ábra. *(a) Lehetséges körút 11 pont esetén. (b) Méhek találta körút egy kaptár és őt etetőpont esetén.*

Mi most nem próbáljuk meg bizonyítani, melyik a lehető legjobb útvonal. Ehelyett egy egyszerű mohó stratégiát vizsgálunk meg.

▦ **Algoritmikus mintázat.** A mohó gondolat a következő: mindig menjünk tovább abba a még meg nem látogatott városba, amelyik az aktuális városhoz a legközelebb van. Ez a módszer egyszerű és természetes, de nem garantálja, hogy a lehető legrövidebb körutat kapjuk. Minden lépésben a pillanatnyilag legjobbnak tűnő döntést hozza meg, és nem vizsgálja meg előre, hogy ez később milyen következményekhez vezet.

A városokat síkbeli pontokként adjuk meg. Az `i`-edik város koordinátái az `x[i]` és `y[i]` tömbcellákban találhatók. Két város távolságát az euklidészi távolsággal számítjuk:

```
sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))
```

A következő programváz a dinamikus tömbök kezelését, a látogatott városok nyilvántartását és az útvonal kiírását mutatja meg. A legközelebbi még meg nem látogatott város megkeresését az olvasóra hagyjuk.

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

int main() {
    int n;
    float *x;
    float *y;
    int *latogatott;

    scanf("%i", &n);

    x = malloc(n * sizeof(float));
    y = malloc(n * sizeof(float));
    latogatott = calloc(n, sizeof(int));

    if (x == NULL || y == NULL || latogatott == NULL) {
        printf("Sikertelen memoriafoglalas.\n");

        free(x);
        free(y);
        free(latogatott);

        return 1;
    }

    for (int i = 0; i < n; i++) {
        scanf("%f %f", &x[i], &y[i]);
    }

    int aktualis = 0;
    int indulasi_varos = aktualis;

    latogatott[aktualis] = 1;

    printf("Utvonal:\n");
    printf("%i", aktualis);

    for (int lepes = 1; lepes < n; lepes++) {
        int legkozelebbi = -1;

        /* Keressuk meg a legkozelebbi meg nem latogatott
        varost */
```

```
            aktualis = legkozelebbi;
            latogatott[aktualis] = 1;

            printf(" %i", aktualis);
        }

        printf(" %i\n", indulasi_varos);

        free(x);
        free(y);
        free(latogatott);

        return 0;
    }
```

A `latogatott` tömb jelzi, hogy mely városokban jártunk már. Kezdetben minden eleme `0`, mert `calloc` segítségével foglaltuk. Amikor a program kiválaszt egy várost, a megfelelő helyre `1` kerül.

A visszaút külön lépés. Az útvonal azért zárul az indulási várossal, mert az utazóügynök körutat tesz meg.

A bemutatott mohó stratégia gyorsan ad egy lehetséges útvonalat, de nem biztos, hogy a lehető legjobbat. Egy közeli város kiválasztása pillanatnyilag jó döntésnek tűnhet, de később előfordulhat, hogy emiatt nagy kerülőt kell tenni.



AI-csepp

Kérdezd meg az AI-t!

- Vezesd végig a mohó algoritmust egy 5 városból álló példán!
- Ha rögzítjük az indulási várost, hány lehetséges körút létezik 5, 10, 15 és 50 város esetén? Mi az általános képlet `n` városra?
- Miért nevezhető mohónak az a stratégia, amely mindig a legközelebbi még meg nem látogatott várost választja?
- Adj példát olyan esetre, amikor a mohó módszer megtalálja a legrövidebb körutat, és olyanra is, amikor nem találja meg!
- Léteznek olyan feladatok, amelyek esetében egy mohó stratégia bizonyíthatóan optimális megoldást ad?
- Hogyan lehetne kiegészíteni az algoritmust, hogy az útvonal hosszát is megadja?

8.7. ECN szál: nagy mátrix feldolgozása két sorral



Egyes versenyfeladatokban nem egy kész mátrixot kapunk bemenetként, hanem a mátrix elemeit valamilyen szabály alapján kell előállítani. Gyakran az is előfordul, hogy a következő sor kiszámításához csak az előző sorra van szükség.

Ilyenkor nem feltétlenül érdemes, sőt nagy méretek esetén nem is lehet a teljes mátrixot eltárolni a memóriában. Ha például egy mátrixnak 100 000 sora és 100 000 oszlopa lenne, akkor az 10 milliárd darab egész számot jelentene. Ez messze túl sok memória lenne egy egyszerű program számára.

Ha azonban a következő sor mindig kiszámítható az aktuális sorból, akkor elég két dinamikusan foglalt egydimenziós tömböt használnunk:

```
aktualis    az éppen feldolgozott sor
kovetkezo   az ebből előállított következő sor
```

Miután a `kovetkezo` sor elkészült, a következő lépésben már ez lesz az aktuális sor. A régi `aktualis` tömböt pedig újra felhasználhatjuk a következő sor kiszámításához. Ehhez nem kell az elemeket átmásolni egyik tömbből a másikba. Elég a két pointert felcserélni. Ezután az `aktualis` pointer már arra a tömbre mutat, amelyikben az imént kiszámított sor található, a `kovetkezo` pointer pedig arra a memóriaterületre, amelyikbe a következő sort írhatjuk.

Nézzünk egy egyszerű példát, az úgynevezett Pascal-háromszöget. Az első sor egyetlen `1`-es elemből áll. Minden következő sor eggyel hosszabb, mint az előző: ha az aktuális sor hossza `m`, akkor a következő sor hossza `m + 1` lesz.

A következő sor elemeit az aktuális sorból az alábbi szabály szerint számítjuk ki:

```
kovetkezo[0] = aktualis[0]

kovetkezo[j] = aktualis[j - 1] + aktualis[j], ha 1 <= j < m

kovetkezo[m] = aktualis[m - 1]
```

Az alábbi program a Pascal-háromszög első `n` sorát generálja (`n > 0`). A következő sor elemeinek az aktuális sorból történő kiszámítását az olvasóra hagytuk.

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

int main() {
    int n;
    int *aktualis;
```

```
    int *kovetkezo;

    scanf("%i", &n);

    aktualis = malloc(n * sizeof(int));
    kovetkezo = malloc(n * sizeof(int));

    if (aktualis == NULL || kovetkezo == NULL) {
        printf("Sikertelen memoriafoglalas.\n");

        free(aktualis);
        free(kovetkezo);

        return 1;
    }

    aktualis[0] = 1;

    printf("%i\n", aktualis[0]);

    int m = 1;

    for (int sor = 2; sor <= n; sor++) {

        /* kovetkezo[0..m] generálása aktualis[0..m - 1]
        alapján */

        for (int j = 0; j <= m; j++) {
            printf("%i ", kovetkezo[j]);
        }
        printf("\n");

        int *seged = aktualis;
        aktualis = kovetkezo;
        kovetkezo = seged;

        m++;
    }

    free(aktualis);
    free(kovetkezo);

    return 0;
}
```

A program a Pascal-háromszög első `n` sorát írja ki, de egyszerre csak két sort tart memóriában. Az `aktualis` tömbben van az a sor, amelyet éppen kiírunk. A `kovetkezo` tömbbe számítjuk ki a következő sort.

A példában `int` típus szerepel az egyszerűség kedvéért, de a Pascal-háromszög elemei gyorsan nagyra nőnek, ezért nagyobb `n` esetén túlcsordulás történhet. Itt azonban nem maga a Pascal-háromszög a lényeg, hanem a memóriahasználati ötlet.

Ha egy feladatban a következő állapot előállítható az aktuális állapotból, akkor gyakran nincs szükség az összes korábbi állapot eltárolására.

▦ **Algoritmikus mintázat.** Ha egy nagy mátrix vagy táblázat következő sora kiszámítható az aktuális sorból, akkor a teljes mátrix helyett elég lehet két egydimenziós tömb:

```
aktualis  -> ebből olvasunk
kovetkezo -> ebbe írunk
```

A számítás után nem másoljuk át az elemeket. Csak a két pointert cseréljük fel:

```
int *seged = aktualis;
aktualis = kovetkezo;
kovetkezo = seged;
```

Ez gyorsabb és memóriatakarékosabb, mint a teljes mátrix tárolása vagy a sorok elemenkénti másolása.

[i] Megjegyzés. A pointerek felcserélése nem az elemeket cseréli ki, hanem azt változtatja meg, hogy melyik név melyik memóriaterületre mutat. Ezért ez a művelet nagyon gyors, független attól, hogy a tömbök hány elemet tartalmaznak.



AI-csepp

Kérdezd meg az AI-t!

- Vezesd végig a program működését `n = 5` esetén!
- Miért kell két tömb, és miért nem elég egyetlen tömb?
- Milyen más feladatokban jelenhet meg az a mintázat, hogy csak az aktuális és a következő sort tároljuk?

►► Zárógondolat

Ebben a fejezetben a változók mögé néztünk: nemcsak az értékükkel foglalkoztunk, hanem azzal is, hol találhatók a memóriában. A pointerek segítségével megtanultunk címeket tárolni, értékeket közvetetten elérni és tömbökön úgy végighaladni, hogy közben nemcsak indexekben, hanem memóriacímekben is gondolkodunk.

A dinamikus memóriafoglalás új lehetőséget adott a programjainknak. A tömb mérete nemcsak a program szövegében szerepelhet, hanem futás közben is eldőlhet. A `malloc`, a `calloc` és a `free` használatával olyan tömböket hozhatunk létre, amelyek mérete a bemenettől függ, és amelyek memóriájáról nekünk kell gondoskodnunk.

A fejezet példái azt is megmutatták, hogy a pointerek nemcsak az adatok elérésére, hanem algoritmikus ötletek megvalósítására is alkalmasak.

A pointerek elsőre szokatlanok lehetnek, mert a programozás egy mélyebb rétegét teszik láthatóvá. De éppen ezért fontosak: segítségükkel nemcsak azt mondjuk meg, milyen értékekkel dolgozik a program, hanem azt is, hogyan jut el ezekhez az értékekhez, hogyan rendezi át őket, és hogyan bánik a rendelkezésére álló memóriával.

9. Mesterfejezet – összekapcsolás és elmélyítés

A könyv előző fejezeteiben tudatosan a leggyakrabban használt nyelvi elemekre szorítkoztunk. A változók esetében elsősorban az `int` és `float` típusokra, a kifejezéseknél az alapvető műveletekre, az utasításkészletben pedig azokra a szerkezetekre koncentráltunk, amelyekkel már a problémák túlnyomó többsége megoldható. Ez összhangban áll az első fejezetben bemutatott Pareto-elvvel: kevés, jól megválasztott eszközzel sokféle feladat kezelhető.

Ez a fejezet erre az alapra épít. Egyrészt a már megismert fogalmakat teljesebb szemléleti keretben tárgyalja, másrészt új nyelvi elemeket is bevezet. A megértést tovább mélyíti azáltal, hogy ezek az elemek nem elszigetelten, hanem egymással szoros kölcsönhatásban jelennek meg.

A mesterfejezet célja nem új szabályok bemagolása, hanem annak megmutatása, hogyan áll össze a korábban megismert nyelvi elemekből egy rugalmasan használható, egységes eszköztár, amely már közelebb áll a valódi programozási feladatok gondolkodásmódjához.

9.1. Tudjunk meg többet a változókról

A C nyelvben egy változó használata a deklarálással kezdődik, amely egyszerre rögzíti a változó *típusát* és *nevét*. A nevet vagy azonosítót az angol ábécé kis- és nagybetűi, számjegyek és aláhúzásjelek alkothatják, kezdőkarakterként azonban csak betű vagy aláhúzásjel szerepelhet, és kulcsszó nem lehet.

A deklaráció nem csupán szintaktikai követelmény: meghatározza, hogy a változó milyen értékeket vehet fel, mennyi memóriát foglal, és milyen műveletek értelmezhetők rajta.

```
<típus> <azonosító>;
```

A deklaráció során a változó kezdőértéket is kaphat, amely meghatározza az első felhasználáskor érvényes állapotát. A *kezdőérték* megadása különösen fontos összetettebb programokban, ahol a változó későbbi viselkedése gyakran a kezdeti állapottól függ.

```
<típus> <azonosító> = <kezdőérték>;
```

Emlékezzünk arra is, hogy egy változó azonosítója egy *memóriacímhez* kötött, ahol az adott típus szerinti érték tárolódik, a memóriacímére pedig az `&` operátorral hivatkozunk.

A 9.1. ábra azt szemlélteti, miként helyezkedhetnek el az `ev` és `pi` változók a memória rekeszeiben. A megadott memóriacímek nyilván fiktívek, és a `0x` előtag azt jelzi, hogy – a megszokott gyakorlatnak megfelelően – hexadecimális formában (azaz 16-os számrendszerben) vannak megadva (lásd lentebb).

```
int ev = 2025;    // 00000000000000000000011111101001
float pi = 3.14;  // 01000000010010001111010111000011

Cím         Tartalom
--------------------
0x1000      11101001 ← ev = 2025
0x1001      00000111
0x1002      00000000
0x1003      00000000
--------------------
0x1004      11000011 ← pi = 3.14
0x1005      11110101
0x1006      01001000
0x1007      01000000
```

9.1. ábra. *A 2025 egész értéket és a 3.14 valós értéket tároló `int`, illetve `float` változók egy lehetséges memóriabeli ábrázolása*

A bájtok sorrendje az adott architektúrától függ. A fenti ábra egy lehetséges elrendezést mutat, és kizárólag szemléltetési célt szolgál: azt mutatja meg, hogy a változók értékei bináris formában, bájtokra bontva kerülnek eltárolásra.

9.1.1. Változók típusa

Alaptípusoknak tekintjük a `char`, `int`, `float` és `double` típusokat, amelyek közül az első kettő egész, míg az utóbbi kettő valós értékek tárolására szolgál. A `char` megnevezés arra utal, hogy gyakran karakterek tárolására használjuk, valójában azonban ezek ASCII kódja kerül eltárolásra, amely egy egész szám (lásd később).

Korábban már szó volt arról, hogy a valós számok lebegőpontos ábrázolás szerint kerülnek bináris formába, innen származik a `float` elnevezés. A `double` kulcsszó arra utal, hogy a `float`-hoz képest nagyobb, jellemzően kétszeres pontosságot biztosít.

Mielőtt ez a készlet szegényesnek tűnne, érdemes megemlíteni a típusmódosítókat, amelyek segítségével további típusok képezhetők. Ezek a `short`, `long`,

`long long`, `signed` és `unsigned` kulcsszavak, amelyek elsősorban az egész típusok értelmezési tartományát és reprezentációját módosítják.

A C nyelvben az egész típusok az alaptípusok és a típusmódosítók kombinációjaként állnak elő. Az alábbiakban azokat az értelmes és szabványos kombinációkat soroljuk fel, amelyek a gyakorlatban is előfordulnak:

- `char`, `unsigned char`
- `short int`, `unsigned short int`
- `int`, `unsigned int`
- `long int`, `unsigned long int`
- `long long int`, `unsigned long long int`

A `signed` módosító az egész típusok esetén alapértelmezettnek tekinthető, ezért gyakran elhagyjuk.

Hasonlóképpen, ha egy deklarációban nem szerepel explicit alaptípus, csak a típusmódosító (`short`, `long`, `long long` vagy `unsigned`), akkor a fordító azt `int`-nek tekinti. Ez a szabály történelmi okokra vezethető vissza, és a nyelv korai változataiból öröklődött. Bár a modern C szabványokban az implicit `int` használata már nem ajánlott, sőt új kódban kerülendő, a szabály ismerete fontos a meglévő kódok megértéséhez és az összetett deklarációk helyes értelmezéséhez.

A lebegőpontos típusok nem kombinálhatók a fenti módosítókkal, ezek önálló típusként jelennek meg:

- `float`
- `double`
- `long double`

Egy típus méretét a `sizeof` operátor segítségével kérdezhetjük le. A C szabvány csak annyit rögzít, hogy:

`sizeof`(char)≤`sizeof`(short)≤`sizeof`(int)≤`sizeof`(`long`)≤`sizeof`(`long long`)

Mivel a tényleges méretek platform- és fordítófüggők lehetnek, egy konkrét táblázat helyett inkább a típusok közötti relációkra és azok szerepére érdemes támaszkodni.

Az egész típusok elsősorban számlálásra, indexelésre és vezérlési feladatokra szolgálnak. Közülük az `int` tekinthető alapértelmezett választásnak: a legtöbb ciklusváltozó, tömbindex és állapotjelző természetes módon írható le ezzel a típussal. Amennyiben az értéktartomány meghaladhatja az `int` által biztosított határokat, a `long long` használata nyújt hordozható megoldást. Az `unsigned` változatok olyan mennyiségek leírására alkalmasak, amelyek értelmezésük szerint nem lehetnek negatívak, például darabszámok vagy méretek.

A lebegőpontos típusok valós számok közelítő ábrázolására szolgálnak. Ezek közül a `double` a leggyakrabban használt típus, mivel megfelelő pontosságot biztosít a legtöbb számítási feladathoz. A `float` elsősorban akkor indokolt, ha a memóriaigény vagy a számítási sebesség kritikus tényező, míg a `long double` speciális, nagy pontosságot igénylő numerikus problémák esetén jelenik meg.

A `char` típus különleges szerepet tölt be: egyszerre használható kis egész értékek tárolására és karakterek kezelésére. Tömbökkel kombinálva (`char[]`) a C nyelvben a szövegek alapvető reprezentációját adja, így jelentősége messze túlmutat egyetlen bájt tárolásán (lásd később).

A C nyelvben a programok túlnyomó többsége az `int`, `double` és `char` típusokra, valamint ezek tömbös változataira épül; a többi típus elsősorban speciális igények kielégítésére szolgál.

[i] Megjegyzés: bár a C szabvány nem rögzíti az egyes típusok pontos méretét, hasznos tisztában lenni azzal, hogy egy adott bájtszám nagyságrendileg milyen értéktartományt vagy pontosságot tesz lehetővé (lásd a 9.1. és 9.2. táblázatokat). Ez segít felmérni, hogy egy probléma esetén egy adott típus elvileg elegendő lehet-e, még akkor is, ha a konkrét megvalósítás platformfüggő.

9.1. táblázat. *Egész típusok – bájtméret és értéktartomány. Ezek az intervallumok a ma gyakorlatilag mindenhol használt 2-es komplemens szerinti ábrázolásra épülnek. Lásd a (Kátai 2008) jegyzet függelékét.*

Méret	Előjeles egész tartomány	Előjel nélküli egész tartomány
1 byte (8 bit)	−128 … 127	0 … 255
2 byte (16 bit)	−32 768 … 32 767	0 … 65 535
4 byte (32 bit)	−2 147 483 648 … 2 147 483 647	0 … 4 294 967 295
8 byte (64 bit)	$-9{,}22 \times 10^{18}$ … $9{,}22 \times 10^{18}$	0 … $1{,}84 \times 10^{19}$

9.2. táblázat. *Lebegőpontos típusok – tipikus pontosságok*

Típus	Méret (tipikusan)	Pontosság
`float`	4 byte	kb. 6–7 decimális számjegy
`double`	8 byte	kb. 15–16 decimális számjegy

[i] Megjegyzés: érdemes még megemlíteni a `size_t` típust, amely egy előjel nélküli egész típus, és amelyet a C nyelv méretek és elemszámok ábrázolására használ. A `sizeof` operátor által visszaadott érték típusa `size_t`, de ugyanezt a típust alkalmazza a nyelv tömbök méretének és memóriaterületek nagyságának kezelésére is. A `size_t` pontos mérete nem rögzített, hanem platform- és fordítófüggő, azonban mindig képes ábrázolni a rendszer által kezelhető legnagyobb objektum méretét. Előjel nélküli volta abból adódik, hogy egy méret vagy elemszám nem lehet negatív.

▶ Összegzés

A változók a C nyelv alapvető építőelemei: egy névhez egy memóriacímet és egy típust rendelnek. A típus nem csupán a tárolható értékek halmazát határozza

meg, hanem azt is, hogy a változón milyen műveletek értelmezhetők. A hordozható programozás szempontjából nem a konkrét bájtméretek, hanem a típusok szerepe és egymáshoz viszonyított relációja a döntő.



AI-csepp
Kérdezd meg az AI-t:
Milyen szempontok alapján döntenél egy változó típusáról, ha a programot több különböző platformon is futtatni kell?

9.2. Konstansok

A konstansok olyan értékek, amelyek a program futása során nem változnak. A C nyelvben többféle konstans létezik, amelyek eltérő alakban jelennek meg a forráskódban, és eltérő szemantikai szerepet töltenek be. Fontos hangsúlyozni, hogy a konstansok is típussal rendelkeznek, még akkor is, ha ezt a típust nem mindig adjuk meg explicit módon.

9.2.1. Egész konstansok

Az egész konstansok alapértelmezés szerint tízes számrendszerben értelmeződnek, de a C nyelv más számrendszerek használatát is lehetővé teszi.

- `0` előtag esetén a szám oktális (8-as) számrendszerű,
- `0x` vagy `0X` előtag esetén hexadecimális (16-os) számrendszerű.

Például:

```
10      // 10 decimálisan
012     // 10 oktálisan
0xA     // 10 hexadecimálisan
```

Az egész konstans típusa a fordító által választott legkisebb olyan egész típus lesz a jelöltek közül, amelyben az érték ábrázolható (tipikusan `int`, nagyobb értéknél `long` vagy `long long`, illetve előjel nélküli változatok). Ezen implicit típus úgynevezett szuffixekkel felülírható. Gyakori szuffixek: `u` vagy `U` (`unsigned`), `l` vagy `L` (`long`), `ll` vagy `LL` (`long long`), illetve ezek kombinációi.

Például:

```
100U          // unsigned int konstans
123456789LL   // long long int konstans
42ULL         // unsigned long long int konstans
```

9.2.2. Valós konstansok

A valós konstansok lebegőpontos értékeket jelölnek, és tartalmazhatnak tizedespontot vagy exponenciális jelölést.

Alapértelmezés szerint egy valós konstans `double` típusú.

Például:

```
3.14    // valós konstans (double)
1e-3    // 0.001 exponenciális alakban (10^-3)
2.0     // valós konstans, double (nem egész!)
12.6E2  // 1260 exponenciális jelölés (12.6 * 10^2)
```

Szuffixekkel a típus itt is módosítható:

- `f` vagy `F`: `float`
- `l` vagy `L`: `long double`

9.2.3. Karakterkonstansok

A karakterkonstansokat aposztrófok (`'`) közé írjuk, és egyetlen karaktert jelölnek. A háttérben ezek is egész értékként, a karakter kódjával kerülnek tárolásra: például `'A'` a kódtáblázat szerinti megfelelő számértéket jelenti. Sok környezetben ez az ASCII-kód (American Standard Code for Information Interchange), amely a gyakori karakterekhez (az angol ábécé betűihez, a számjegyekhez és az írásjelekhez) 0 és 127 közötti egész számokat rendel. Ez magyarázza, hogy a karakterkonstansok miért kezelhetők egész értékként, és miért értelmezhetők rajtuk összehasonlítások is.

Közvetlenül leírható karakterkonstansok:

```
'A'    // nagybetű
'7'    // számjegy
'?'    // írásjel
' '    // szóköz
```

Hasznos tudni, hogy az ASCII kódtáblában a számjegyek kódjai 48-tól, a nagybetűk kódjai 65-től, a kisbetűk kódjai pedig 97-től kezdődnek.

Nem minden karakter írható le közvetlenül aposztrófok között. A nem megjeleníthető vezérlőkaraktereket escape-szekvenciákkal adjuk meg, amelyek szintén karakterkonstansok. Íme néhány példa:

```
'\n'   // újsor (ASCII 10)
```

```
'\t'   // tabulátor (ASCII 9)
'\0'   // null karakter, karakterlánclezáró (ASCII 0)
'\\'   // backslash (ASCII 92)
'\''   // aposztróf (ASCII 39)
```

A kódolásfüggő vagy nehezen megadható karakterekre numerikus alakban is hivatkozhatunk, a karakter kódjának megadásával:

```
'\x41' // 'A' hexadecimális kóddal (ASCII 65)
'\101' // 'A' oktális kóddal (ASCII 65)
```

9.2.4. Karakterlánc-konstansok

A karakterlánc-konstansokat idézőjelek (") határolják, és karakterek sorozatát jelentik. A C nyelvben ezek nullával lezárt karaktertömbként (`char[]`) jelennek meg a memóriában.

Például:

```
"Hello"      // öt karakter + lezáró '\0'
""           // üres karakterlánc, csak a lezáró '\0'
"Szoveg\n"   // szöveg újsor karakterrel a végén
"%i %i"      // formátum-karakterlánc (printf/scanf)
"\"hello\""  // idézőjel karakterek a karakterláncban
```

Fontos különbség, hogy míg egy karakterkonstans egyetlen egész érték, addig egy karakterlánc memóriában tárolt adatsor, amely több egymást követő címen helyezkedik el.

[i] Megjegyzés: hosszú karakterlánc-konstansok több sorba is tördelhetők a sorvégi \ karakter segítségével. Emellett az egymás után írt karakterlánc-konstansokat a fordító automatikusan egyetlen karakterlánccá fűzi össze.

9.2.5. Szimbolikus konstansok

A C nyelv lehetőséget ad szimbolikus konstansok definiálására, amelyek egy nevet rendelnek egy értékhez. Ez történhet, például, a `#define` direktivával vagy a `const` kulcsszó használatával.

Például:

```
#define PI 3.141592    // általában az #include-ok után
const int MAX = 100;   // igazából írásvédett változó
```

A szimbolikus konstansok növelik a program olvashatóságát, csökkentik a „mágikus számok” számát, és megkönnyítik az értékek későbbi módosítását.

Beépített (szabványos) szimbolikus konstansok: a C szabványkönyvtár több fejlécfájlban is biztosít előre definiált határértékeket és jellemzőket. Ezek különösen akkor hasznosak, amikor hordozható módon szeretnénk hivatkozni egy típus tartományára vagy pontosságára.

- `<limits.h>`: egész típusok határértékei (pl. INT_MAX, INT_MIN, LLONG_MAX)
- `<float.h>`: lebegőpontos típusok jellemzői (pl. DBL_MAX, DBL_EPSILON)
- `<stdint.h>`: fix méretű egész típusok és határértékeik (pl. int32_t, INT32_MAX)

9.2.6. Lebegőpontos csapdák

A DBL_EPSILON használatára egy hasznos példa az úgynevezett lebegőpontos csapdák elkerülése. A lebegőpontos számok nem minden valós számot tudnak pontosan ábrázolni. Ennek következménye, hogy látszólag egyszerű műveletek is meglepő eredményt adhatnak.

Például:

```
double x = 0.1 + 0.2;
if (x == 0.3) {
    /* ide általában nem jutunk el */
}
```

Ennek oka, hogy a 0.1 és a 0.2 értékek bináris alakban csak közelítőleg ábrázolhatók, így az összegük sem lesz pontosan 0.3.

Helyes megközelítés: lebegőpontos értékeket mindig egy tűréshatáron belül hasonlítsunk össze. (Ahogyan az `abs` függvény egész számok esetén, úgy az `fabs` függvény lebegőpontos értékek esetén adja meg a paraméter abszolút értékét, `double` típusú eredményként.)

```
if (fabs(x - 0.3) < DBL_EPSILON) {
    /* x gyakorlatilag 0.3 */
}
```

Tanulság: a lebegőpontos számokkal végzett számításoknál nem az a kérdés, hogy két érték egyenlő-e, hanem az, hogy elég közel vannak-e egymáshoz.

▶ **Összegzés:** a konstansok a C nyelvben nem pusztán „beégetett számok”, hanem típusos, szemantikailag jelentéssel bíró elemek. Tudatos használatuk

hozzájárul az átláthatóbb, biztonságosabb és könnyebben karbantartható programok készítéséhez.



AI-csepp

Kérd meg az AI-t:

- Véleményezd a következő állítást: „Nincs negatív egész konstans a C-ben.”
- Véleményezd a következő állítást: „A `const` által deklarált konstansok valójában írásvédett változók.”
- Add meg a vezérlőkarakterek teljes listáját escape-szekvenciákkal!

9.3. Formátumkarakterek a `scanf` és `printf` használatában

A változók és konstansok után természetes következő lépés annak megértése, hogyan kapcsolódnak ezek a program külső világához. A `scanf` és a `printf` formátumkarakter-láncai pontosan ezt a hidat jelentik: leírják, hogy a memóriában tárolt értékeket milyen típus szerint kell értelmezni, illetve hogyan kell azokat beolvasni vagy megjeleníteni.

A korábbi fejezetekben csak az `%i` és `%f` formátumkarakterekkel dolgoztunk, mivel a példák az `int` és `float` típusokra korlátozódtak. Most, hogy megismertük a változótípusok jóval teljesebb készletét, szükségessé válik a formátumkarakterek körének bővítése is.

A 9.3.–9.6. táblázatok a formátumkaraktereket típuscsoportok szerint rendezik.

9.3. táblázat. *Karaktertípus*

Típus	`printf`	`scanf`	Megjegyzés
`char`	`%c`	`%c`	egyetlen karakter; nem hagyja ki a whitespace-t

9.4. táblázat. *Egész típusok*

Típus	`printf`	`scanf`	Megjegyzés
`short int`	`%hi`	`%hi`	rövid előjeles egész
`unsigned short int`	`%hu`	`%hu`	rövid előjel nélküli
`int`	`%i`	`%i`	alapértelmezett egész
`unsigned int`	`%u`	`%u`	előjel nélküli
`long int`	`%li`	`%li`	hosszú egész

Típus	printf	scanf	Megjegyzés
unsigned long int	%lu	%lu	hosszú előjel nélküli
long long int	%lli	%lli	nagyon nagy értéktartomány
unsigned long long int	%llu	%llu	előjel nélküli nagy értéktartomány

Ahogyan a konstansoknál láttuk, hogy egész értékeket megadhatunk tízes, nyolcas és tizenhatos (hexadecimális) számrendszerben, ugyanígy a be- és kimeneti műveleteknél is lehetőség van arra, hogy az adatokat különböző számrendszerek szerint értelmezzük vagy jelenítsük meg. A scanf a formátumkarakterek alapján dönti el, hogy a bemeneten érkező karaktereket milyen számrendszer szerint alakítsa egész értékké, míg a printf ugyanezekkel a formátumkarakterekkel szabályozza, hogy egy egész érték milyen alakban kerüljön kiírásra.

Egész típusok esetén az alábbi formátumkarakterek használhatók:

- %i – egész szám automatikus számrendszer-felismeréssel (10, 012, 0xA alakok egyaránt elfogadottak a scanf esetén),
- %d – egész szám decimális (10-es) alakban,
- %o – egész szám oktális (8-as) alakban,
- %x – egész szám hexadecimális alakban, kisbetűs számjegyekkel,
- %X – egész szám hexadecimális alakban, nagybetűs számjegyekkel.

Ezek a formátumkarakterek tehát nem új típusokat jelölnek, hanem azt határozzák meg, hogy ugyanazt az egész értéket milyen számrendszer szerint kell értelmezni (scanf) vagy megjeleníteni (printf). Ez a szemlélet közvetlenül összekapcsolja a konstansoknál megismert számrendszer-fogalmat az I/O műveletek működésével.

Példa:

```
scanf("%i", &x);
  // automatikus felismerés: 10, 012 vagy 0xA alak is elfogadott
printf("%d %o %X\n", x, x, x);
  // háromféle kiírás ugyanarról az értékről
```

Példa:

```
int a, b;

scanf("%i", &a);   // ha a bemenet: 012 → oktális, értéke: 10
scanf("%d", &b);   // ha a bemenet: 012 → decimális, értéke: 12

printf("%d %d\n", a, b);  // 10 12
```

9.5. táblázat. *Lebegőpontos típusok*

Típus	printf	scanf	Megjegyzés
`float`	`%f`	`%f`	`printf`-ben `double`-ként adódik át
`double`	`%f`	`%lf`	`scanf`-nél kötelező az `l`
`long double`	`%Lf`	`%Lf`	nagy pontosság

9.6. táblázat. *Karakterláncok és memóriacím típus*

Típus	printf	scanf	Megjegyzés
`char[]`	`%s`	`%s`	`scanf` nem ellenőrzi a tömb méretét
memóriacím (mutató/pointer)	`%p`	`%p`	memóriacím megjelenítése

9.3.1. Formázott kiírás – mezőszélesség és pontosság

A `printf` formátumkarakterei nemcsak az érték típusát határozzák meg, hanem annak megjelenési módját is. A formátumkarakter elé írt szám a mezőszélességet adja meg: azt, hogy a kiírt érték legalább hány karakterhelyet foglaljon el.

```
printf("%4i", x);
```

Ebben a formátumban a 4 azt jelenti, hogy az egész érték legalább 4 karakter széles mezőben jelenik meg. Ha az érték kevesebb karaktert igényel, a `printf` bal oldalon szóközökkel tölti ki a mezőt; ha több karakter szükséges, a mező automatikusan bővül.

Ez a fajta igazítás különösen hasznos táblázatszerű kimenetek esetén, ahol az adatok oszlopokba rendezve jelennek meg.

Lebegőpontos értékeknél a formátumkarakter két számot is tartalmazhat:

```
printf("%8.2f", y);
```

Itt a 8 a mezőszélességet jelöli (legalább 8 karakter), a .2 a pontosságot, vagyis a tizedesjegyek számát.

Például az `3.14159` érték az alábbi alakban jelenik meg. Öt, azaz 8-2-1 karakter széles mezőben jelenik meg az egész rész, ami azt jelenti, hogy a 3-as érték elé 4 szóköz kerül.

```
    3.14
```

A mezőszélesség itt is az igazítást szolgálja, míg a pontosság szabályozza a megjelenített numerikus részletességet.

▶ **Összegzés**

Összegző példa – változók, beolvasás és kiírás formátumkarakterekkel.

```
int i;
scanf("%i", &i);              // %i: automatikus számrendszer
printf("%d\n", i);            // kiírás decimális alakban

unsigned int u;
scanf("%u", &u);              // előjel nélküli egész beolvasása
printf("%u\n", u);            // előjel nélküli egész kiírása

long long ll;
scanf("%lli", &ll);           // nagy tartományú egész beolvasása
printf("%lli\n", ll);         // nagy egész kiírása

double d;
scanf("%lf", &d);             // valós szám beolvasása
printf("%f\n", d);            // valós szám kiírása

char c;
scanf("%c", &c);              // egyetlen karakter beolvasása
printf("%c\n", c);            // karakter kiírása

char s[100];                  // a tömbök neve egyben a címük is
scanf("%s", s);               // karakterlánc beolvasása, nincs &
printf("%s\n", s);            // karakterlánc kiírása

printf("%p\n", &i);           // memóriacím kiírása
```

A formátumkarakter-lánc nem puszta segédszöveg, hanem szerződés a program és a futtatókörnyezet között: rögzíti, hogy az adatok milyen típus szerint kerülnek értelmezésre. Ennek a szerződésnek a megsértése gyakran nem fordítási hibát, hanem nehezen felderíthető futásidejű problémákat okoz.



AI-csepp

Kérdezd meg az AI-t:

- A fentiekben többször találkoztunk a platformfüggő/platformfüggetlen, hordozható/nem hordozható, illetve szabványos/nem szabványos kifejezésekkel. Mit jelentenek ezek pontosan a C nyelv kontextusában?

- Miért nem hordozható a `printf("%lu", sizeof(int));` kifejezés, és mi a helyes, szabványos megoldás helyette?

9.4. Tömbök

9.4.1. Egydimenziós tömbök rövid áttekintése

A 6. fejezetben már bevezettük az egydimenziós tömbök fogalmát. Ezek deklarálásának általános szintaxisa:

```
<típus> <név>[<méret>];
```

A 7.6. alfejezetben arra is láttunk példát, hogyan adhatunk kezdőértékeket a tömb elemeinek már a deklaráláskor. Ennek általános alakja:

```
<típus> <név>[<méret>] = {<kezdőérték_lista>};
```

Az alábbi példák az egydimenziós tömbök inicializálásának legfontosabb eseteit mutatják be:

```
int t1[10] = {0,1,1,2,3,5,8,13,21,34};
// pontosan 10 elem, mindegyik explicit módon megadva

int t2[10] = {0,1,1,2,3,5,8};
// az első 7 elem kap értéket, a maradék 3 automatikusan 0 lesz

int t3[10] = {0,1,1,2,3,5,8,13,21,34,55};
// HIBÁS: túl sok kezdőérték (11 érték 10 elemhez)

int t4[1000] = {0};
// minden elem 0-ra inicializálódik

int t5[1000] = {1};
// csak az első elem lesz 1, a többi 0

int t6[] = {0,1,1,2,3,5,8,13,21,34};
// a tömb méretét a fordító határozza meg (itt 10 elem)

int t7[];
// HIBÁS: a tömb mérete nem ismert, és nincs kezdőérték
```

9.4.2. Kétdimenziós tömbök

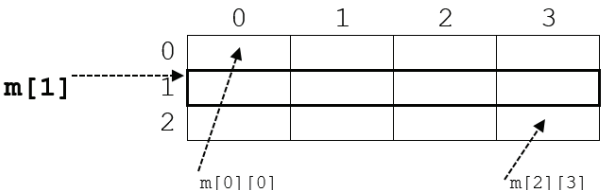
Az egydimenziós tömbök természetes általánosításaként a C nyelvben kétdimenziós tömbök is használhatók. Deklarálásuk általános alakja:

```
<típus> <név>[<sorok_száma>][<oszlopok_száma>];
```

Ez egy olyan tömböt hoz létre, amely `<sorok_száma>` darab egydimenziós tömböt tartalmaz, és mindegyik sor `<oszlopok_száma>` elemű. Ez a tulajdonság teszi lehetővé, hogy egy kétdimenziós tömb sorait egydimenziós tömbként kezeljük.

Egy kétdimenziós tömb egy elemére két index segítségével hivatkozunk. Ha csak a sorindexet adjuk meg, akkor az adott sorra hivatkozunk. Az indexelés minden dimenzióban 0-tól kezdődik itt is:

```
<név>[<sor_index>][<oszlop_index>]
<név>[<sor_index>]
```



9.2. ábra. *Szemléltető ábra az `int m[3][4];` tömbdeklarációhoz. Az ábrán jelöltük, hogyan hivatkozunk a bal felső (`m[0][0]`) és a jobb alsó (`m[2][3]`) elemre, valamint azt is, hogy a kétdimenziós tömb `m[1]` sora önmagában egydimenziós tömbként értelmezhető.*

[i] Megjegyzés. Bár a kétdimenziós tömböket gyakran mátrixként értelmezzük, ezek nem mátrixok, hanem olyan tárolók, amelyek mátrixokat tárolhatnak. Egy kétdimenziós tömb egymás után tárolt sorokból áll, vagyis egydimenziós tömbök sorozata. Erre mondjuk, hogy a memória sorfolytonosan van lefoglalva. A 9.2. ábrán bemutatott `m` tömb számára $3 \cdot 4 \cdot$ `sizeof(int)` bájt folytonos memóriaterület kerül lefoglalásra.

A kétdimenziós tömbök inicializálása tipikusan beágyazott kapcsos zárójelekkel történik, ahol minden belső kapcsos zárójel egy-egy sort reprezentál:

```
int m1[2][3] = {
    {1, 2, 3},   // első sor
    {4, 5, 6}    // második sor
};
```

Ebben a példában `m1[0][0]` értéke 1, míg `m1[1][2]` értéke 6.

A belső kapcsos zárójelek használata nem kötelező, de erősen ajánlott. A C nyelv megengedi az ún. lapos inicializálást is:

```
int m2[2][3] = {1, 2, 3, 4, 5, 6};
// a fordító sorfolytonosan tölti fel a tömböt
```

Ez ekvivalens a fenti, olvashatóbb formával, amely egyértelműen jelöli a sorhatárokat.

[i] Megjegyzés. Ahogyan egydimenziós tömböknél, kétdimenziós tömböknél is megengedett a részleges kezdőértékadás. Ugyanazok az alapelvek érvényesek itt is: ha kevesebb értéket adunk meg, akár egyes sorokon belül is, akkor a többi elem automatikusan nullával inicializálódik. Ennek leggyakoribb esete, amikor a teljes tömböt nullával szeretnénk feltölteni. Kétdimenziós tömböknél azonban egy fontos sajátos szabály is érvényes: az oszlopok számát mindig explicit módon meg kell adni, mivel a fordítónak ismernie kell egy sor méretét az elemek memóriacímének kiszámításához.

9.4.2.1. Kétdimenziós tömbök feldolgozása

A kétdimenziós tömbök feldolgozása tipikusan egymásba ágyazott `for` ciklusokkal történik.

Az alábbi példa egy 3×4 méretű mátrix beolvasását mutatja be egy azonos méretű kétdimenziós tömbbe, majd ezt követi a tömb tartalmának kiírása táblázatos formában, formázott kimenettel. A `%4i` használatánál megelőlegeztük, hogy az egyes elemek legfeljebb három számjegyűek, így a negyedik karakterhely az igazítást szolgálja.

```
int a[3][4];

for (int i = 0; i < 3; i++) {          // sorok bejárása
    for (int j = 0; j < 4; j++) {      // oszlopok bejárása
        scanf("%i", &a[i][j]);         // egyetlen elem beolvasása
    }
}

for (int i = 0; i < 3; i++) {          // sorok
    for (int j = 0; j < 4; j++) {      // oszlopok
        printf("%4i", a[i][j]);        // egy elem kiírása igazítva
    }
    printf("\n");                      // új sor minden sor végén
}
```

Az egymásba ágyazott ciklusszerkezet révén az (i, j) indexpárok az

(0,0), (0,1), (0,2), (0,3), (1,0), (1,1), (1,2), (1,3), (2,0), (2,1), (2,2), (2,3)

sorrendben generálódnak, ami biztosítja a kétdimenziós tömb soronkénti, celláról cellára történő bejárását.

Egy $n \times m$ méretű mátrix eltárolható egy $n \cdot m$ elemű egydimenziós tömbben is ($a[0..n \cdot m-1]$). Amennyiben a feldolgozás logikája megköveteli, a kétdimenziós indexelés bármikor visszaállítható. Az alábbi példa elemzését az olvasóra hagyjuk.

```
for (int i = 0; i < n; i++) {          // sorok
    for (int j = 0; j < m; j++) {      // oszlopok
        scanf("%i", &a[i * m + j]);    // (i, j) → lineáris
index
    }
}

for (int k = 0; k < n * m; k++) {
    int i = k / m;       // sorindex
    int j = k % m;       // oszlopindex

    int x = a[k];        // a mátrix (i, j) eleme
    /* ... feldolgozás (i,j) pozíciójú elemként ... */
}
```

Ez a megközelítés világosan megmutatja, hogy a kétdimenziós tömbök szintaxisa mögött valójában lineáris memóriakezelés áll. A különbség csupán az indexelésben van: az (i, j) indexpár az i*m+j képlettel képezhető le egyetlen lineáris indexre, míg fordítva a sor- és oszlopindex a k/m és k%m műveletekkel állítható vissza.

9.4.2.2. Szimmetriák négyzetes mátrixokban

Kétdimenziós tömbökben tárolt négyzetes mátrixok hatékony feldolgozásánál gyakran hasznos az alapvető szimmetriák figyelembevétele. Tekintsük a t[0..n-1][0..n-1] alakú $n \times n$ mátrixot, ahol egy elem pozícióját az (i, j) indexpár jelöli.

A t[i][j] elem szimmetrikus pozíciói:

- főátlóra ($i = j$) tükrözés: $(i, j) \rightarrow (j, i)$
- mellékátlóra ($i + j = n - 1$) tükrözés: $(i, j) \rightarrow (n\text{-}1\text{-}j, n\text{-}1\text{-}i)$
- középpontra (180°-os elforgatás): $(i, j) \rightarrow (n\text{-}1\text{-}i, n\text{-}1\text{-}j)$

Főátló és a két háromszög

- főátló: $i = j$

• főátló felett: $j > i$
• főátló alatt: $j < i$

A főátlóra szimmetrikus elemek páronként lefedik egymást, ezért sok algoritmus elég, ha csak az egyik háromszöget dolgozza fel.

Mellékátló és a két háromszög
• mellékátló: $i + j = n - 1$
• mellékátló felett: $i + j < n - 1$
• mellékátló alatt: $i + j > n - 1$

Pásztázási minták

1) A főátlón fentről lefelé

```
for (int i = 0; i < n; i++) {       // i: 0..n-1
    int x = t[i][i];                // főátló eleme: (i, i)
    /* ... feldolgozás ... */
}
```

2) A főátló feletti elemek bejárása soronként, fentről lefelé, és egy soron belül balról jobbra

```
for (int i = 0; i < n; i++) {          // sorok
    for (int j = i + 1; j < n; j++) { // j > i: főátló felett
        int x = t[i][j];               // felső háromszögben
        /* ... feldolgozás ... */
    }
}
```

3) A mellékátló alatti elemek pásztázása oszloponként, jobbról balra, és egy oszlopon belül lentről felfelé

```
for (int j = n - 1; j >= 0; j--) {  // oszlopok jobbról balra
    for (int i = n - 1; i > n - 1 - j; i--) {  // i + j > n-1
        int x = t[i][j];    // mellékátló alatti háromszögben
        /* ... feldolgozás ... */
    }
}
```

A mátrixban az átlók gyakran egyszerűen leírhatók állandó kifejezéssel.
• Főátlóval párhuzamos átlókra jellemző, hogy $i - j = konstans$; a főátló maga: $i - j = 0$.
• Mellékátlóval párhuzamos átlókra jellemző, hogy $i + j = konstans$; a mellékátló maga: $i + j = n - 1$.

Ezek az invariánsok különösen hasznosak, amikor egy algoritmus egy adott átlót vagy átlósávot szeretne feldolgozni.

9.4.3. Többdimenziós tömbök (*n*-dimenziós általánosítás)

A kétdimenziós tömbök természetes módon általánosíthatók több dimenzióra is. A C nyelvben elvileg tetszőleges számú dimenzió megadható.

Például az

```
int t[2][3][4];
```

deklaráció nyomán egy háromdimenziós tömb jön létre, amely 2 darab, egyenként 3×4 méretű kétdimenziós tömböt tartalmaz.

Általános alakban egy `k` dimenziós tömb deklarációja:

```
<típus> <név>[d1][d2]...[dk];
```

Egy többdimenziós tömb egy elemére annyi indexszel hivatkozunk, ahány dimenziója van:

```
<név>[i1][i2]...[ik]
```

Szemléletileg egymásba ágyazott tömbök sorozatáról beszélhetünk: minden dimenzió egy újabb strukturális szintet jelent.

Természetesen minden többdimenziós tömb is sorfolytonosan kerül tárolásra. Ez azt jelenti, hogy a legutolsó index változik a leggyorsabban, a többi dimenzió pedig ennek megfelelően „csomagolja” az adatokat.

Egy

```
int t[d1][d2][d3];
```

alakú háromdimenziós tömb `t[i][j][k]` elemének memóriabeli helye az alábbi gondolat mentén számítható ki:

```
eltolás = i · (d2 · d3) + j · d3 + k
```

Ez a képlet a korábban megismert kétdimenziós eset közvetlen általánosítása.

[i] Megjegyzés. Bár a C nyelv támogatja az *n*-dimenziós tömböket, a gyakorlatban a két- és háromdimenziós tömbök a leggyakoribbak. Nagyobb dimenziószám esetén az adatok kezelése és a kód olvashatósága gyorsan romlik, ezért ilyenkor gyakran egydimenziós tömböt és explicit indexelést alkalmazunk.



AI-csepp

A korábbi fejezetekben találkoztunk változó méretű tömbökkel, például az alábbi programrészletben:

```
int n;
scanf("%i", &n);
int a[n];
```

Kérdezd meg az AI-t!

- Miért nem ajánlott a változó méretű tömbök (VLA – Variable Length Array) használata a C nyelvben, még akkor sem, ha a fordító támogatja őket?
- Miért kell a kétdimenziós tömbök oszlopméretét mindig ismernie a fordítónak?

9.5. A pointerek haladóbb használata

A 8. fejezetben megismerkedtünk a pointerek alapgondolatával. Láttuk, hogy egy pointer egy objektum címét tárolja, a címképző `&` operátorral lekérdezhetjük egy objektum címét, a dereferáló `*` operátorral pedig elérhetjük a megcímzett értéket. Megvizsgáltuk azt is, hogyan kapcsolódnak a pointerek az egydimenziós tömbökhöz, hogyan működik a pointeraritmetika, valamint hogyan foglalhatunk dinamikusan memóriát a `malloc` és a `calloc` függvényekkel.

Ebben a részben ezekre az ismeretekre építünk. Először azt vizsgáljuk meg, hogyan változtatható meg egy pointer típus szerinti értelmezése. Ezután megismerkedünk az általános célú `void *` pointerrel és a `const` minősítő pointerekre gyakorolt hatásával. Végül az egydimenziós tömböknél látott tömb–pointer kapcsolatot kétdimenziós tömbökre terjesztjük ki, majd bemutatjuk a dinamikusan létrehozott kétdimenziós adatszerkezetek legfontosabb megvalósítási módjait.

9.5.1. Pointerek explicit típuskonverziója

Korábban hangsúlyoztuk, hogy a pointer típusa meghatározza, milyen típusú objektumként értelmezi a fordító a megcímzett memóriaterületet. Egy `int *` típusú pointer dereferálásakor `int` értéket kapunk, és a pointer eggyel történő növelése egy `int` objektumnyi lépést jelent.

A különböző pointertípusok nem cserélhetők fel tetszőlegesen. Bizonyos helyzetekben azonban szükség lehet arra, hogy egy pointerértéket más pointertípusként kezeljünk. Ezt explicit típuskonverzióval, vagyis casttal tehetjük meg.

A cast általános alakja:

```
(<céltípus>) <kifejezés>
```

Pointer esetén például:

```
int x = 1999;
int *ip = &x;
unsigned char *cp = (unsigned char *)ip;
```

A cast nem alakítja át az `x` objektumot, és nem módosítja a memóriában található adatot. Csupán azt változtatja meg, hogy a fordító milyen típusú pointerként kezeli ugyanazt a címet.

Ez különösen jól látható a pointeraritmetikán:

```
int x;
int *ip = &x;
unsigned char *cp = (unsigned char *)&x;

ip + 1;   /* egy int objektumnyi lépés */
cp + 1;   /* egy bájtnyi lépés */
```

A két pointer kezdetben ugyanazt a címet reprezentálja, de eltérő típusuk miatt eltérő lépésmérettel működnek.

A pointer castolása önmagában nem garantálja, hogy az új pointer biztonságosan dereferálható. Ehhez több feltételnek is teljesülnie kell:

- a címnek megfelelően igazítottnak kell lennie a céltípushoz;
- elegendő memóriának kell rendelkezésre állnia;
- a megcímzett objektumot a C szabályai szerint megengedett módon kell elérnünk.

A következő kód ezért nem tekinthető egy egész szám szabályos `float` típusú értelmezésének:

```
int x = 1999;
float *fp = (float *)&x;

/* printf("%f\n", *fp); */
```

Még akkor sem biztonságos a dereferálás, ha az adott rendszeren az `int` és a `float` mérete egyaránt négy bájt. A cast elhallgattathat egy fordítói figyelmeztetést, de nem tesz érvényessé egy megkérdőjelezhető memória-hozzáférést. Erre biztonságos megoldást később látunk majd.

Egy objektum bájtszintű reprezentációja karaktertípusú pointeren keresztül vizsgálható meg:

```
int x = 1999;
```

```
unsigned char *bytes = (unsigned char *)&x;

for (size_t i = 0; i < sizeof(x); ++i) {
    printf("%02x ", bytes[i]);
}
```

Ebben az esetben nem változtatjuk meg az objektum típusát, csupán az őt alkotó bájtokat vizsgáljuk meg.

9.5.2. A `void *` pointer

Az egydimenziós dinamikus tömböknél már használtuk a `malloc` és a `calloc` függvényeket. Ezek visszatérési típusa `void *`. A `void *` olyan általános objektumpointer, amely bármilyen objektumtípus címét képes tárolni.

```
int x = 2011;
void *vp = &x;
```

A `void *` pointerben megőrződik a cím, de a megcímzett objektum konkrét típusa a pointer típusából már nem állapítható meg. Emiatt a `void *` nem dereferálható közvetlenül:

```
void *vp = &x;

/* printf("%d\n", *vp); */   /* hibás */
```

A dereferálás előtt a pointert megfelelő objektumpointer-típusként kell kezelnünk:

```
int *ip = vp;
printf("%d\n", *ip);
```

C nyelvben az objektumpointerek és a `void *` közötti konverzió automatikus, ezért az explicit cast nem szükséges:

```
int *ip = (int *)vp;   /* helyes, de C-ben felesleges */
```

Ugyanezért az egydimenziós dinamikus tömb foglalásánál sem szükséges castolni a `malloc` eredményét:

```
int *a = malloc(n * sizeof *a);
```

A `void *` pointeren szabványos C-ben pointeraritmetika sem végezhető:

```
void *vp = &x;

/* vp++; */   /* nem szabványos C */
```

A fordító ugyanis nem tudja, mekkora objektumnyi lépést kellene végrehajtania. Ha bájtonként szeretnénk haladni, előbb karaktertípusú pointerre kell váltanunk:

```
unsigned char *p = vp;
p++;
```

A `void *` fő szerepe az általános interfészek kialakítása. Olyan függvények használhatják, amelyeknek többféle objektumtípussal kell együttműködniük. Az általánosság ára az, hogy a programozónak kell nyilvántartania, milyen típusú objektum található ténylegesen a címen.

9.5.3. A `const` és a pointerek

Korábban láttuk, hogy egy pointeren keresztül nemcsak olvashatjuk, hanem módosíthatjuk is a megcímzett objektumot:

```
int x = 10;
int *p = &x;

*p = 20;
```

A `const` minősítővel külön szabályozhatjuk, hogy módosítható-e a megcímzett objektum a pointeren keresztül, illetve módosítható-e maga a pointer.

Pointer csak olvasható objektumhoz

```
int x = 2017;
int y = 2018;

const int *p = &x;
```

A `p` pointer másik objektumra átállítható, de a megcímzett érték `p` pointeren keresztül nem módosítható:

```
/* *p = 2023; */   /* hibás */
```

```
p = &y;             /* megengedett */
```

A `const` ebben az esetben az adott hozzáférési útvonalra vonatkozik. Maga az `x` objektum nem feltétlenül konstans:

```
x = 2023;           /* megengedett */
```

Konstans pointer módosítható objektumhoz

```
int *const p = &x;
```

Itt maga a pointer nem állítható át másik címre, de a megcímzett objektum módosítható:

```
*p = 2023;          /* megengedett */
/* p = &y; */       /* hibás */
```

Mivel a pointer értéke nem változhat, a konstans pointert a deklarációjakor inicializálni kell.

Konstans pointer csak olvasható objektumhoz

```
const int *const p = &x;
```

Ebben az esetben sem maga a pointer, sem a megcímzett objektum nem módosítható a pointeren keresztül:

```
/* *p = 2023; */    /* hibás */
/* p = &y; */       /* hibás */
```

A három alak összefoglalva:

```
const int *p1;            /* változó pointer, csak olvasható int
*/
int *const p2 = &x;       /* konstans pointer, módosítható int */
const int *const p3 = &x; /* konstans pointer, csak olvasható int
*/
```

9.5.4. Kétdimenziós tömbök és pointerek

Az egydimenziós tömböknél láttuk, hogy a tömb neve a legtöbb kifejezésben az első elemére mutató pointerré alakul. Egy kétdimenziós tömb esetén ugyanez az elv érvényesül, de a tömb első eleme maga is egy tömb.

```
int matrix[3][4];
```

A `matrix` három elemből álló tömb. Minden eleme egy négy `int` értéket tartalmazó tömb.

Ezért a `matrix` kifejezés a legtöbb helyzetben nem `int *`, hanem egy négyelemű `int` tömbre mutató pointer:

```
int (*p)[4] = matrix;
```

A deklarációt a változó nevétől kifelé olvashatjuk: `p` pointer egy négy `int` elemet tartalmazó tömbhöz.

A kétdimenziós indexelés pointerműveletekre vezethető vissza:

```
matrix[i][j]
```

jelentése:

```
*(*(matrix + i) + j)
```

A `matrix + i` az `i`-edik sorra mutat. A külső dereferálás kiválasztja ezt a sort, majd a `+ j` és a második dereferálás kiválasztja a sor `j`-edik elemét.

Például:

```
int matrix[3][4] = {
    { 1,  2,  3,  4},
    { 5,  6,  7,  8},
    { 9, 10, 11, 12}
};

printf("%d\n", matrix[1][2]);
printf("%d\n", *(*(matrix + 1) + 2));
```

Mindkét kifejezés a 7 értéket adja.

A sorra mutató pointer növelése egy teljes sorral lép tovább:

```
int (*p)[4] = matrix;
```

```
p++;   /* a következő, négyelemű sorra lép */
```

Ez eltér egy `int *` pointer növelésétől:

```
int *q = matrix[0];

q++;   /* a sor következő int elemére lép */
```

A típus itt is meghatározza a lépésméretet.

9.5.5. Dinamikusan létrehozott kétdimenziós tömbök

Az egydimenziós dinamikus tömböknél egyetlen összefüggő memóriaterületet foglaltunk, amelynek kezdőcímét pointerben tároltuk. Kétdimenziós adatok esetén többféle megvalósítás közül választhatunk.

Egyetlen összefüggő memóriablokk

A legegyszerűbb megoldás, hogy `rows * cols` darab elemet foglalunk:

```
size_t rows = 3;
size_t cols = 4;

int *a = malloc(rows * cols * sizeof *a);

if (a == NULL) {
    /* sikertelen foglalás */
}
```

Ebben az esetben az elemeket egydimenziós indexeléssel érjük el:

```
a[i * cols + j]
```

Például:

```
for (size_t i = 0; i < rows; ++i) {
    for (size_t j = 0; j < cols; ++j) {
        a[i * cols + j] = 0;
    }
}
```

A memória összefüggő, ezért egyetlen `free` hívással felszabadítható:

```
free(a);
```

Ez a megoldás egyszerű és hatékony, de az elemek eléréséhez kézzel kell kiszámítani az indexet.

Dinamikus tömb változó hosszúságú sorokra mutató pointerrel

Ha az oszlopszám futás közben ismert, foglalhatunk teljes sorokat tartalmazó összefüggő blokkot is:

```
size_t rows = 3;
size_t cols = 4;

int (*a)[cols] = malloc(rows * sizeof *a);

if (a == NULL) {
    /* sikertelen foglalás */
}
```

Itt `a` egy `cols` darab `int` elemet tartalmazó sorra mutató pointer. Emiatt a megszokott kétdimenziós indexelés használható:

```
a[i][j] = 0;
```

A `sizeof *a` egy teljes sor méretét adja, ezért a foglalás `rows` darab sornak biztosít helyet.

A memória ebben az esetben is összefüggő, és egyetlen hívással felszabadítható:

```
free(a);
```

Ez a megoldás jól tükrözi a valódi kétdimenziós tömb típusát, megtartja az `a[i][j]` jelölést, és egyetlen memóriablokkot használ.

Soronként külön foglalt kétdimenziós szerkezet

Egy másik lehetőség, hogy először a sorokra mutató pointerek tömbjét foglaljuk le:

```
int **a = malloc(rows * sizeof *a);

if (a == NULL) {
    /* sikertelen foglalás */
}
```

Ezután minden sort külön foglalunk:

```
for (size_t i = 0; i < rows; ++i) {
    a[i] = malloc(cols * sizeof *a[i]);

    if (a[i] == NULL) {
        /* az addig lefoglalt sorokat fel kell szabadítani */
    }
}
```

Az elemek ebben az esetben is elérhetők az ismerős alakban:

```
a[i][j]
```

A sorok azonban nem feltétlenül helyezkednek el egymás mellett a memóriában.

A felszabadítást fordított sorrendben kell elvégezni:

```
for (size_t i = 0; i < rows; ++i) {
    free(a[i]);
}

free(a);
```

A soronkénti foglalás előnye, hogy az egyes sorok hossza különbözhet.

A megoldások összehasonlítása

Az egyetlen összefüggő blokk: `int *a`;
- egyszerűen foglalható és szabadítható fel;
- a memória összefüggő;
- az indexet `i * cols + j` alakban kell kiszámítani.

A sorra mutató pointer: `int (*a)[cols]`;
- összefüggő memóriát használ;
- támogatja az `a[i][j]` indexelést;
- a sorhossz a pointer típusának része.

A pointerek pointere: `int **a`;
- minden sort külön memóriablokkban tárolhat;
- támogatja az eltérő sorhosszakat;
- több foglalást és több felszabadítást igényel;
- nem kompatibilis egy valódi kétdimenziós tömbre mutató pointerrel.

Fontos tehát, hogy az `int **` nem általános helyettesítője az `int [rows][cols]` típusú tömbnek. A két reprezentáció memóriaszerkezete és pointertípusa eltér.

▶ Összegzés

A pointerek típusa nem puszta formai információ. Meghatározza, hogyan értelmezi a fordító a megcímzett objektumot, milyen típusú értéket ad a dereferálás, és mekkora lépést jelent a pointeraritmetika.

A cast megváltoztatja a pointer típus szerinti értelmezését, de nem alakítja át automatikusan a memóriában található objektumot. A `void *` általános objektumpointerként használható, de dereferálás előtt ismernünk kell a tényleges objektumtípust. A `const` segítségével külön szabályozhatjuk a pointer és a megcímzett objektum módosíthatóságát.

Kétdimenziós tömb esetén a tömb neve egy teljes sorra mutató pointerré alakul. A dinamikus kétdimenziós adatok megvalósíthatók egyetlen lineáris blokkal, sorokra mutató pointerrel vagy külön lefoglalt sorokkal. A megfelelő megoldás kiválasztásakor figyelembe kell venni a memória összefüggőségét, az indexelés módját, a sorok méretét és a felszabadítás összetettségét.



AI-csepp

Kérdezd meg az AI-t!

- Milyen feltételek mellett dereferálható biztonságosan egy más típusra castolt pointer?
- Miért nem helyes egy `int` objektumot `float *` pointeren keresztül kiolvasni, még akkor sem, ha a két típus azonos méretű?
- Miben különbözik a `void *` használata C-ben és C++-ban?
- Mi történik, ha casttal eltávolítjuk a `const` minősítést?
- A `int matrix[3][4];` deklarálás nyomán mi a különbség a `matrix`, a `matrix[0]`, az `&matrix[0]` és az `&matrix[0][0]` kifejezések típusa között?
- Milyen előnye van az összefüggő memóriafoglalásnak?

▶▶ Zárógondolat

A fejezet során látszólag különálló elemekkel foglalkoztunk: típusokkal, konstansokkal, tömbökkel, formátumkarakterekkel és mutatókkal. A közös nevező azonban végig ugyanaz volt: hogyan reprezentálódnak az adatok a memóriában, és hogyan értelmezi őket a C nyelv.

A C nem elrejti a memória működését, hanem láthatóvá és befolyásolhatóvá teszi azt. Ez egyszerre ad nagy szabadságot és nagy felelősséget a programozónak. A fejezet célja nem az volt, hogy minden részletet lezárjon, hanem hogy kialakítson egy olyan gondolkodásmódot, amelyben az érték, a cím és a típus egységet alkot.

Erre a szemléletre épül a következő mesterfejezet, amely már nem az adatok szerkezetére, hanem a vezérlésre és a program futásának logikájára koncentrál.

10. Mesterfejezet: kifejezések és vezérlés – a C valódi arca

Az előző mesterfejezet az adatok világára koncentrált: arra, hogyan reprezentálja a C nyelv az értékeket a memóriában, és miként kapcsolódnak össze az egyszerű típusok, a tömbök és a mutatók. Ez a szemlélet megmutatta, mit kezel a program.

Egy program működését azonban nemcsak az adatok, hanem a vezérlési döntések is meghatározzák. A C nyelvben ebben a kifejezések töltenek be kitüntetett szerepet: önálló utasításként is megjelenhetnek, miközben a döntési és ismétlési szerkezetek vezérlését is kifejezések adják meg.

Ez a mesterfejezet a kifejezések és a vezérlési szerkezetek működését vizsgálja emeltebb szinten. A cél nem az egyes szerkezetek újbóli bemutatása, hanem annak megértése, hogyan alkotnak egységes logikai rendszert. A fejezet külön figyelmet szentel azoknak a pontoknak, ahol a C nagy szabadsága könnyen félreértéshez vagy hibás programhoz vezethet.

A fejezet végére az olvasó képes lesz a vezérlést nem külső irányításként, hanem a program belső logikájaként értelmezni. Ezen a ponton válik igazán láthatóvá a C nyelv valódi arca.

10.1. Kifejezések

A C nyelvben a számítások és döntések alapegysége a kifejezés. Egy kifejezés operátorokkal összekapcsolt operandusokból áll, és mindig rendelkezik értékkel és típussal. Ez akkor is igaz, ha a kifejezés első ránézésre „utasításnak” tűnik. A C nyelv sajátossága, hogy az értékadás is kifejezés, nem pusztán utasítás.

10.1.1. Operandusok és operátorok

Egy kifejezés operandusai lehetnek:

- változók,
- konstansok,
- más kifejezések,
- függvényhívások.

Például a másodfokú egyenlet valós gyökeit meghatározó kifejezés utasításokban mind a négyféle operandus megtalálható:

```
x1 = (-b + sqrt(d)) / (2*a);
```

Az operátorok pedig meghatározzák, hogy ezek között milyen művelet történik. A korábban megismert aritmetikai (`+ - * / %`), összetett értékadó (`+= -= *= /= %=`), inkrementáló/dekrementáló (`++ --`), relációs (`< <= > >= == !=`) és logikai (`&& || !`) operátorok, valamint indexelési (`[]`), címképző (`&`), dereferáló (`*`), `sizeof` és az explicit típuskonverziót megvalósító cast operátor csak egy részét adják a teljes készletnek. Ebben az alfejezetben a már ismert operátorokról további részleteket tudhat meg az olvasó, és természetesen eddig még nem tárgyalt operátorok is bemutatásra kerülnek.

A C-ben az operátoroknak aritása van:

- unáris operátor: egy operandus (pl. `-x`, `!flag`, `++i`, `sizeof(int)`)
- bináris operátor: két operandus (pl. `a + b`, `x < y`)
- ternáris operátor: három operandus (`?:`)

10.1.2. Operátorprecedencia és zárójelezés

A kifejezések kiértékelési sorrendjét az operátorprecedencia és az asszociativitás határozza meg. Bár ezek táblázatba foglalhatók, a gyakorlatban egy egyszerű szabály követése ajánlott: ha kétség merül fel, használjunk zárójeleket.

```
a + b * c    // b * c előbb (operátorprecedencia)
(a + b) * c  // egyértelmű szándék (zárójelezés)
a + b + c    // balról jobbra értékelődik (asszociativitás)
int *t[10]   // t 10 elemű tömb: elemei int * típusúak
int (*p)[10] // p pointer: 10-elemű int-tömb címét tárolhatja
```

Az utolsó két példa az illusztrálja, hogy a `[]` operátornak nagyobb a prioritása, mint a `*` operátornak. Másfelől, a zárójelezés gyakran nemcsak a fordítónak, hanem az olvasónak is szól.

10.1.3. A kifejezések értéke és típusa

Minden kifejezésnek van egy értéke, amit a futás során képvisel, és egy típusa, amely meghatározza az értelmezést és a további műveleteket. Bár a kifejezések típusa nem mindig „látszik” a forráskódban, a fordító minden esetben meghatározza.

Például:

```
3 + 4       // értéke: 7, típusa: int
3.0 + 4     // értéke: 7.0, típusa: double
'a' + 1     // értéke: 'b' kódja, típusa: int
```

10.1.3.1. Implicit típuskonverzió

Ahogy a fenti példákból is kiderült, ha egy bináris operátor két operandusa különböző típusú, akkor automatikusan implicit típuskonverzióra kerül sor.

A C nyelvben az implicit típuskonverzió nem egyetlen lépésben történik. Először az ún. egészpromóciók lépnek életbe: a `char` és `short` típusú operandusok `int`-té (vagy `unsigned int`-té) konvertálódnak, mielőtt bármilyen aritmetikai művelet végrehajtásra kerülne. Ezt követik az ún. szokásos aritmetikai konverziók, amelyek az operátorok operandusait közös típusra hozzák. Általános elv, hogy az „alacsonyabb rangú” típus felfelé konvertálódik (ha van lebegőpontos, minden lebegőpontos irányba megy, különben a kisebb értéktartományú igazodik a nagyobb értéktartományúhoz), és az eredmény típusa a művelet szempontjából „nagyobb rangú”, kiegyenlített típus lesz.

További példák:

```
int i = 3;
double d = 2.5;
i + d    // i → double, eredmény: double

char c = 'A';
c + 1    // c → int, eredmény: int
```

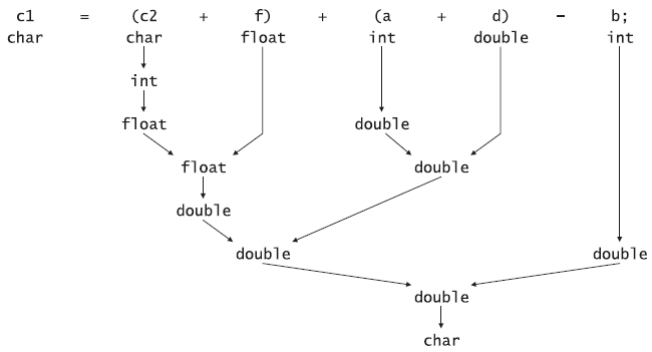
Az értékadás esetén azonban külön szabály érvényesül: a jobb oldali kifejezés értéke a bal oldali változó típusára konvertálódik, ezt nevezzük értékadási konverziónak.

```
int i;
double d = 3.7;

i = d;    // értékadási konverzió: d → int, i értéke 3
```

Ebben az esetben a jobb oldali lebegőpontos érték az értékadás során egész típussá konvertálódik. A konverzió információvesztéssel járhat, de a C nyelv ezt megengedi, és nem tekinti hibának.

A 10.1. ábra egy összetett kifejezés kiértékelése során a háttérben zajló implicit típuskonverziókat szemlélteti.



10.1. ábra. *Implicit típuskonverziók egy összetett kifejezés kiértékelése során*

10.1.3.2. Explicit típuskonverzió

Ha az implicit konverzió nem felel meg a kívánt szemantikának, explicit típusátalakítást alkalmazhatunk.

Emlékezzünk például arra, hogy a `(double)i/j` kifejezés garantálja, hogy a művelet lebegőpontos osztásként történjen, még akkor is, ha `i` és `j` egyébként egészek. Ha egy lebegőpontos változó tárolta valós szám egész részre van szükségünk, akkor is explicit típuskonverziót használhatunk, például `int` típusra.

A cast mindig egy unáris operátor, amely egy új, ideiglenes értéket hoz létre. Nem módosítja az eredeti változót.

10.1.4. Aritmetikai operátorok

Az aritmetikai operátorokról részletesen beszéltünk a 3. fejezetben, ugyanígy az összetett aritmetikai értékadó operátorokról is. Itt néhány további részletet közlünk az inkrementáló és dekrementáló operátorokkal kapcsolatban.

10.1.4.1. Inkrementáló és dekrementáló operátorok

A ++ és -- operátorok unáris operátorok, de kétféle alakban fordulhatnak elő:

```
++i   // prefix
i++   // postfix
```

A különbség kifejezésekben jelentkezik:

```
x = ++i;   // i nő, majd x megkapja az új értéket
x = i++;   // x megkapja a régi értéket, majd i nő
```

Ez a különbség tehát csak akkor releváns, ha az operátor egy nagyobb kifejezés része.

10.1.5. Relációs és logikai kifejezések

Az összehasonlító és logikai operátorokkal az olvasó már találkozhatott az 5. fejezetben; most ezek szerepét és működését kiegészítjük és pontosítjuk.

Bár az összehasonlító operátorok (`< <= > >= == !=`) eredményét logikai természetűnek (igaz vagy hamis) érzékeljük, valójában egész értéket adnak, amelyhez az igaz vagy hamis logikai értelmezés társul. Ugyanez igaz a logikai operátorokra (`&& || !`) is, amelyek gyakran ezekből építkeznek.

Ennek hátterében az áll, hogy a C nyelvben a numerikus és logikai értékek között kölcsönös megfeleltetés létezik. Konkrétan: minden nem nulla numerikus érték igaznak számít. A logikai kifejezések eredménye a gyakorlatban jellemzően 1 az igaz és 0 a hamis esetén, de a szabvány csak azt garantálja, hogy az igaz érték nem nulla.

Ez magyarázza meg, hogy az alábbi `if` feltételkifejezések ekvivalensek:

```
if (x != 0) {...}
if (x) {...}
```

Hasznos továbbá tudni, hogy a `&&` és `||` operátorok „rövidzárasak”:

- `A && B`: ha `A` hamis, `B` nem kerül kiértékelésre,
- `A || B`: ha `A` igaz, `B` nem kerül kiértékelésre.

Ez nemcsak optimalizáció, hanem gyakran biztonsági mechanizmus is:

```
if (i < n && a[i] > 0) {
    ...
}
```

Itt garantált, hogy `a[i]` csak akkor kerül kiértékelésre, ha `i` érvényes index.

[i] Megjegyzés: De Morgan-szabályok. A C nyelvben gyakran szükség van összetett feltételek átalakítására. Ilyenkor hasznos eszközt adnak a De Morgan-szabályok, amelyek megmutatják, hogyan osztható szét a negálás (logikai tagadás) összetett logikai kifejezéseken:

```
!(A && B)  ≡  (!A || !B)
!(A || B)  ≡  (!A && !B)
```

Ezek a szabályok nemcsak matematikailag helyesek, hanem a gyakorlatban is segítik az olvashatóbb és hibamentesebb feltételek megfogalmazását.

Tipikus C-s helyzet például, hogy egy

```
if (!(a > 0 && b < 10)) { ... }
```

alakú feltétel az átalakítás után így írható fel:

```
if (a <= 0 || b >= 10) { ... }
```

[i] Megjegyzés: logikai típus (`bool`) a C nyelvben. A C nyelv klasszikus modelljében nincs önálló logikai alaptípus: az igaz és hamis értékeket egész számok képviselik (0: hamis, nem nulla: igaz). A C99 szabvány óta azonban a `<stdbool.h>` fejléc egy kényelmi típust vezet be, a `bool`-t, amely ezt a szemantikát nevekkel látja el: a `true` és `false` szimbólumok rendre 1 és 0 értékeket jelölnek.

Szemléltető példa (feltételezi a `#include <stdbool.h>` kódsort):

```
int befizetes;
bool rendezett = false;

scanf("%i", &befizetes);

if (befizetes >= 1000) {
    rendezett = true;
}

if (rendezett) {
    printf("A tartozás rendezve van.\n");
} else {
    printf("A befizetés nem elegendő.\n");
}
```

A `bool` típus használata nem vezet be új szabályokat, csupán olvashatóbbá és önmagyarázóbbá teszi a kódot. Használata modernizálja a forráskódot, és hidat épít más, újabb nyelvek (például C++, Java, Python) logikai modellje felé, miközben a C nyelv alapvető működését változatlanul hagyja.

10.1.6. A feltételes operátor (?:)

A C nyelv egyetlen ternáris (három operandusú) operátora a feltételes operátor. Általános alakja:

```
<feltétel> ? <kifejezés1> : <kifejezés2>
```

A <feltétel> kiértékelése után, ha annak értéke igaz (nem nulla), akkor a teljes kifejezés értéke a <kifejezés1> értéke lesz, ellenkező esetben a <kifejezés2> értéke. A két kifejezés közül csak az egyik kerül kiértékelésre, hasonlóan a logikai && és || operátorok rövidzáras működéséhez.

A feltételes operátor nem vezérlési szerkezet, hanem kifejezés, ezért közvetlenül értékadásban is használható:

```
int absz = (x >= 0) ? x : -x;
```

Ez a forma gyakran tömörebb és kifejezőbb, mint egy azonos hatású if-else szerkezet, ugyanakkor túlzott vagy egymásba ágyazott használata rontja az olvashatóságot.

10.1.7. A vessző operátor (,)

A vessző operátor két (vagy több) kifejezést kapcsol össze egyetlen kifejezéssé. Kiértékelése balról jobbra történik: először a bal oldali kifejezések értékelődnek ki (ezek értékei elvesznek), majd a jobb oldali, és a teljes kifejezés értéke a jobb oldali kifejezés értéke lesz.

```
x = (a = 1, b = 2, a + b);
```

A fenti kifejezés hatására a értéke 1, b értéke 2, x pedig 3 lesz.

A vessző operátor tipikusan olyan helyzetekben jelenik meg, ahol a szintaxis egyetlen kifejezést vár, de több műveletet szeretnénk végrehajtani. Klasszikus példája egy olyan for ciklus, amely egy tömb két végéről indítva két ciklusváltozót léptet egymással szemben, amíg azok nem találkoznak:

```
for (i = 0, j = n - 1; i < j; i++, j--) {
    /* ... */
}
```

Fontos megkülönböztetni a vessző operátort azoktól a vesszőktől, amelyek deklarációkban vagy függvényhívások argumentumlistájában szerepelnek. Utóbbiak nem operátorok, csupán szintaktikai elválasztók.

10.1.8. Az értékadás mint kifejezés

A C egyik legfontosabb – és sokszor félreértett – tulajdonsága, hogy az értékadás kifejezés, amelynek értéke maga a hozzárendelt érték.

```
x = 5         // értéke: 5
y = (x = 5)   // x = 5, majd y = 5
y = x = 5
  /* az előzővel ekvivalens, mert az értékadások
     jobbról balra sorrendben hajtódnak végre */
```

[i] Megjegyzés: bal- és jobbérték (l-value és r-value). A C nyelvben nem minden kifejezés használható az értékadás bal oldalán. Azokat a kifejezéseket, amelyek egy konkrét memóriacímhez tartoznak, és érték adható nekik, balértéknek nevezzük (például egy változó vagy egy tömb egy eleme). A jobbértékek ezzel szemben ideiglenes értékek, amelyek nem rendelkeznek önálló memóriacímmel, ezért nem szerepelhetnek értékadás bal oldalán. Az értékadás bal oldalán mindig balértéknek kell állnia, míg a jobb oldalon tetszőleges kifejezés szerepelhet. Maga az értékadás is kifejezés, amelynek értéke a hozzárendelt jobbérték.

```
y = (x = 5);
// x balérték
// (x = 5) jobbértékként viselkedik
// y balérték
// a teljes értékadás kifejezésnek van értéke, ezért beágyazható
```

▶ Összegzés

A kifejezések a C nyelv „ragasztóanyagai”: ezek kapcsolják össze a változókat, konstansokat, operátorokat és vezérlési szerkezeteket. Egy kifejezés mindig értékkel és típussal rendelkezik, és gyakran önállóan is teljes értelmű építőelem. A C ereje – és egyben veszélye – abban rejlik, hogy a nyelv kevés korlátozást állít: a helyes és olvasható kód megírása ezért tudatos szemléletet igényel.



AI-csepp

Kérdezd meg az AI-t!

- Azt mondtuk, hogy „a kifejezések kiértékelési sorrendjét az operátorprecedencia és az asszociativitás határozza meg”. Jelenti-e ez azt, hogy „ha ismerem a precedenciát, akkor ismerem a kiértékelési sorrendet is”? Fejtsd ki a választ, és adj példát!
- A kifejezések fentebbi bemutatását végigkísérte a mellékhatás mint jelenség, bár sehol nem mondtuk ki. Definiáld a mellékhatás fogalmát a kifejezések kontextusában! Adj példákat!
- A fejezet bevezetőjében előrevetítettük, hogy „a C nagy szabadsága könnyen félreértéshez vagy hibás programhoz vezethet”. Hogyan igaz ez a kifejezésekkel kapcsolatban?

10.2. Bitműveletek és bitmaszkok

A bitműveletek formálisan ugyan a C nyelv operátorai közé tartoznak, ezért kifejezések részeként használjuk őket, tárgyalásuk mégis indokolt egy önálló alfejezetben. Míg az aritmetikai, relációs és logikai operátorokat elsősorban az operandusok számértéke vagy logikai jelentése alapján értelmezzük, a bitműveleti operátorok az egész értékek belső, bináris ábrázolásának egyes bitjein hajtanak végre műveleteket. Megértésükhöz ezért nemcsak az operátorok működését, hanem a kettes számrendszert, a bitszélességet, az előjeles és előjel nélküli típusok sajátosságait, valamint a bitmaszkok használatát is figyelembe kell venni. Önálló tárgyalásuk lehetővé teszi, hogy ezek az összefüggések egységesen, a tipikus gyakorlati alkalmazásokkal együtt jelenjenek meg.

A bitműveleti operátorok használata különösen gyakori alacsony szintű programozásban, hardverközeli alkalmazásokban, kommunikációs protokollokban, jogosultságok és állapotjelzők tárolásakor, valamint olyan esetekben, amikor több logikai információt szeretnénk egyetlen egész értékben tömören ábrázolni.

A C nyelv bitműveleti operátorai:

- bitenkénti ÉS: &
- bitenkénti VAGY: |
- bitenkénti kizáró VAGY: ^
- bitenkénti negálás: ~
- balra tolás: <<
- jobbra tolás: >>

A bitműveleti operátorok csak egész típusú operandusokon alkalmazhatók. Lebegőpontos értékeken közvetlenül nem használhatók.

10.2.1. Bitenkénti logikai operátorok

A `&`, `|` és `^` bináris operátorok a két operandus azonos helyi értékű bitjeit páronként vizsgálják meg.

A bitenkénti ÉS (`&`) eredményében egy bit akkor lesz 1, ha mindkét operandus megfelelő bitje 1.

```
unsigned int a = 12;  // binárisan: 1100
unsigned int b = 10;  // binárisan: 1010

unsigned int c = a & b;  // 1000, azaz 8
```

A művelet bitenként:

```
  1100
& 1010
```

```
------
  1000
```

A bitenkénti VAGY (|) eredményében egy bit akkor lesz 1, ha a két operandus megfelelő bitje közül legalább az egyik 1.

```
unsigned int c = a | b;  // 1110, azaz 14
```

A bitenkénti kizáró VAGY (^) eredményében egy bit akkor lesz 1, ha a két megfelelő bit különbözik.

```
unsigned int c = a ^ b;  // 0110, azaz 6
```

A ^ operátort gyakran XOR operátornak nevezzük az angol *exclusive OR* elnevezés alapján.

Fontos megkülönböztetni a bitenkénti és a logikai operátorokat:

```
a & b    // bitenkénti ÉS
a && b   // logikai ÉS

a | b    // bitenkénti VAGY
a || b   // logikai VAGY
```

A bitenkénti operátorok minden biten elvégzik a műveletet, és az eredményük egy egész érték. A logikai operátorok ezzel szemben az operandusok igaz vagy hamis jellegét vizsgálják, és logikai eredményt adnak.

A bitenkénti & és | operátorok nem rövidzárasak: mindkét operandusuk minden esetben kiértékelődik.

10.2.2. Bitenkénti negálás

A ~ unáris operátor az operandus minden bitjét az ellenkezőjére változtatja: a 0 bitekből 1, az 1 bitekből pedig 0 lesz.

```
unsigned char x = 12;   // 00001100
unsigned char y = ~x;   // 11110011
```

Az eredmény pontos értéke az operandus típusának bitszélességétől függ. Emiatt a bitműveleteknél különösen fontos figyelembe venni, hogy az adott típus hány biten tárolódik.

A ~ operátort nem szabad összetéveszteni a logikai negálást végző ! operátorral:

```
~x   // minden bitet megfordít
!x   // 1, ha x nulla; különben 0
```

Például:

```
unsigned char x = 12;

~x   // nyolcbites ábrázolást feltételezve: 243
!x   // 0, mert x nem nulla
```

A kisebb egész típusú operandusokra a művelet előtt egészpromóció történik. Emiatt például egy `unsigned char` típusú érték negálásakor a `~` művelet ténylegesen rendszerint `int` típuson hajtódik végre. Ha csak az eredeti típus bitjeire van szükségünk, az eredményt vissza kell alakítani vagy ilyen típusú változóban kell eltárolni.

```
unsigned char x = 12;
unsigned char y = (unsigned char)~x;
```

10.2.3. Biteltoló operátorok

A << és >> operátorok az operandus bitjeit balra, illetve jobbra mozgatják. A balra tolás általános alakja:

```
érték << eltolás
```

Például:

```
unsigned int x = 5;     // 00000101
unsigned int y = x << 2; // 00010100, azaz 20
```

A bitek két pozícióval balra tolódnak, a jobb oldalon felszabaduló helyekre pedig nullák kerülnek.

```
00000101 << 2 = 00010100
```

Ha az érték és az eredmény ábrázolható, egy nem negatív egész szám `n` bittel történő balra tolása rendszerint megfelel a 2 n-edik hatványával való szorzásnak:

```
x << n
```

megfelelhet ennek:

```
x * 2^n
```

Ez az összefüggés azonban csak akkor használható biztonságosan, ha az eredmény belefér a típus értéktartományába. Előjeles egész típusok túlcsordulását és negatív értékek balra tolását kerülni kell. Bitműveletekhez ezért általában előjel nélküli egész típusok használata ajánlott.

A jobbra tolás például:

```
unsigned int x = 20;     // 00010100
unsigned int y = x >> 2; // 00000101, azaz 5
00010100 >> 2 = 00000101
```

Előjel nélküli érték jobbra tolásakor a bal oldalon felszabaduló bithelyekre nullák kerülnek. Egy előjel nélküli érték `n` bittel történő jobbra tolása megfelel a `2 n`-edik hatványával végzett egész osztásnak:

```
x >> n
```

megfelel ennek:

```
x / 2^n
```

A negatív előjeles számok jobbra tolásának pontos viselkedése megvalósításfüggő lehet. A hordozható programokban ezért a biteltolásokat célszerű előjel nélküli típusokon végezni.

Az eltolás mértéke nem lehet negatív, és kisebbnek kell lennie a bal oldali operandus típusának bitszélességénél. Ellenkező esetben a program viselkedése nem definiált.

```
unsigned int x = 1;

x << 3;   // helyes
x << -1;  // hibás: negatív eltolás
```

10.2.4. Bitmaszkok

A bitmaszk olyan egész érték, amelynek kijelölt bitjei segítségével más értékek bizonyos bitjeit vizsgálhatjuk vagy módosíthatjuk.

Egy adott bithez tartozó maszk legegyszerűbben úgy állítható elő, hogy az 1 értéket a kívánt pozícióba toljuk:

```
1u << pozicio
```

A számozás a legkisebb helyi értékű bittől indul, amelynek pozíciója 0.

```
1u << 0   // 00000001
1u << 1   // 00000010
1u << 2   // 00000100
1u << 3   // 00001000
```

Az u utótag azt jelzi, hogy az 1 konstans típusa `unsigned int`. Bitműveleteknél ez segít elkerülni az előjeles típusokból eredő problémákat.

Egy bit bekapcsolásához bitenkénti VAGY műveletet használhatunk:

```
ertek = ertek | maszk;
```

Rövidebb alakban:

```
ertek |= maszk;
```

Például a harmadik pozíciójú bit bekapcsolása:

```
unsigned int ertek = 0;
unsigned int maszk = 1u << 3;

ertek |= maszk;
```

Az eredmény:

```
00000000
00001000
--------
00001000
```

Egy bit kikapcsolásához a maszk negáltját és bitenkénti ÉS műveletet használunk:

```
ertek &= ~maszk;
```

Például:

```
unsigned int ertek = 15;   // 00001111
unsigned int maszk = 1u << 2;

ertek &= ~maszk;           // 00001011, azaz 11
```

Egy bit átbillentéséhez, vagyis 0-ról 1-re, illetve 1-ről 0-ra változtatásához XOR művelet használható:

```
ertek ^= maszk;
unsigned int ertek = 8;    // 00001000
unsigned int maszk = 1u << 3;

ertek ^= maszk;            // 00000000
ertek ^= maszk;            // 00001000
```

Egy bit állapotának vizsgálatához bitenkénti ÉS műveletet használhatunk:

```
if (ertek & maszk) {
    /* a kijelölt bit be van kapcsolva */
}
```

A művelet eredménye akkor nem nulla, ha az `ertek` és a `maszk` legalább egy közös 1 bitet tartalmaz.

Ha kifejezetten 0 vagy 1 értéket szeretnénk kapni, a kiválasztott bitet a legalsó pozícióba tolhatjuk, majd 1-gyel maszkolhatjuk:

```
unsigned int bit = (ertek >> pozicio) & 1u;
```

10.2.5. Több állapot tárolása egyetlen értékben

Egy egész érték különböző bitjei önálló állapotjelzőként is használhatók. Az egyes állapotokhoz különböző maszkokat rendelhetünk.

```
#define OLVASHATO  (1u << 0)
#define IRHATO     (1u << 1)
#define FUTTATHATO (1u << 2)
```

Ezek binárisan:

```
OLVASHATO   00000001
IRHATO      00000010
FUTTATHATO  00000100
```

Több jogosultság összekapcsolható a bitenkénti VAGY operátorral:

```
unsigned int jogosultsag = OLVASHATO | IRHATO;
```

Egy jogosultság ellenőrzése:

```
if (jogosultsag & IRHATO) {
    printf("Az eroforras irhato.\n");
}
```

Új jogosultság hozzáadása:

```
jogosultsag |= FUTTATHATO;
```

Jogosultság eltávolítása:

```
jogosultsag &= ~IRHATO;
```

Több megadott bit egyidejű ellenőrzéséhez az eredményt a teljes maszkkal kell összehasonlítani:

```
unsigned int szukseges = OLVASHATO | IRHATO;

if ((jogosultsag & szukseges) == szukseges) {
    printf("Mindket jogosultsag rendelkezesre all.\n");
}
```

A következő feltétel ezzel szemben csak azt vizsgálja, hogy a felsorolt bitek közül legalább az egyik be van-e kapcsolva:

```
if (jogosultsag & szukseges) {
    printf("Legalabb egy jogosultsag rendelkezesre all.\n");
}
```

10.2.6. Összetett bitműveleti értékadások

A bitműveleti operátorokhoz is tartoznak összetett értékadó alakok:

```
x &= y;     // x = x & y;
x |= y;     // x = x | y;
x ^= y;     // x = x ^ y;
x <<= n;    // x = x << n;
x >>= n;    // x = x >> n;
```

Ezek ugyanazt az elvet követik, mint az aritmetikai összetett értékadó operátorok. Előnyük nem pusztán a rövidebb írásmód: a bal oldali kifejezés csak egyszer kerül kiértékelésre.

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

Ebben a kifejezésben az `i++` csak egyszer hajtódik végre.

10.2.7. Precedencia és zárójelezés

A bitműveleti operátorok precedenciája több esetben alacsonyabb, mint elsőre várnánk. Különösen veszélyes lehet a bitműveleti és az összehasonlító operátorok keverése.

Például:

```
if (ertek & maszk == 0) {
    /* ... */
}
```

Ez nem feltétlenül úgy értelmeződik, ahogy az olvasó elsőre gondolná, mert az `==` operátor precedenciája nagyobb a bitenkénti `&` operátorénál. A kifejezés ezért ennek megfelelően csoportosul:

```
ertek & (maszk == 0)
```

A kívánt alak:

```
if ((ertek & maszk) == 0) {
    /* a kijelölt bitek egyike sincs bekapcsolva */
}
```

Bitműveleteknél ezért különösen ajánlott a szándékot zárójelekkel egyértelművé tenni.

Hasonlóképpen:

```
unsigned int eredmeny = (a << 4) | b;
```

A zárójelek akkor is javíthatják az olvashatóságot, ha a precedenciaszabályok alapján nem feltétlenül szükségesek.

[i] Megjegyzés. A klasszikus C szabványokban az egész konstansok decimális, oktális és hexadecimális alakban írhatók fel. A hexadecimális alak különösen kényelmes bitmaszkok megadására, mert minden hexadecimális számjegy pontosan négy bitnek felel meg:

```
0x01u   // 00000001
0x0Fu   // 00001111
0xF0u   // 11110000
0xFFu   // 11111111
```

A bináris `0b...` alakot egyes fordítók korábban is kiterjesztésként támogatták, a C23 szabvány pedig már szabványos formában is bevezette. Régebbi nyelvi szabványt célzó programokban azonban a hexadecimális jelölés hordozhatóbb lehet.

[i] Megjegyzés. A `printf` függvénynek hagyományosan nincs szabványos bináris formátumjelölője. Egy érték bitjeit ciklussal jeleníthetjük meg:

```
#include <stdio.h>

void bitek_kiirasa(unsigned int ertek){
    int i;

    for (i = (int)(sizeof(ertek) * 8) - 1; i >= 0; --i) {
        putchar((ertek & (1u << i)) ? '1' : '0');
    }
}
```

A függvény minden bitpozícióhoz létrehoz egy maszkot, majd megvizsgálja, hogy az adott bit be van-e kapcsolva.

▶ Összegzés

A bitműveleti operátorok az egész értékek bináris ábrázolásának egyes bitjein dolgoznak. A `&`, `|`, `^` és `~` operátorok biteket vizsgálnak vagy módosítanak, míg a `<<` és `>>` operátorok a biteket balra vagy jobbra tolják. Bitmaszkok segítségével egyedi bitek és több bites mezők kapcsolhatók be, törölhetők, átbillenthetők és olvashatók ki.

A bitműveletek tömör és hatékony megoldásokat tesznek lehetővé, de helyes használatukhoz pontosan ismerni kell az operandusok típusát, bitszélességét, az egészpromóciók szabályait és az operátorok precedenciáját. A jól olvasható,

hordozható kód érdekében általában előjel nélküli típusok, beszédes nevű maszkok és egyértelmű zárójelezés használata ajánlott.



AI-csepp
Kérdezd meg az AI-t!
• Hogyan viselkednek a bitműveletek előjeles egész típusok esetén?
• Van-e lehetőség bit szinten hozzáférni lebegőpontos adatokhoz?
• Magyarázd el lépésről lépésre, mit csinál következő kódsor: `ertek |= 1u << 3;`
• A következő két kifejezés mindig ugyanazt az eredményt adja? `x * 4` vs. `x << 2`
• Írj programot, amely megszámolja egy `unsigned int` érték 1-es bitjeit!
• Mikor előnyösebb bitmaszkot használni több különálló logikai változó helyett?
• Miért nem rövidzáras a `&` és a `|` operátor?
• Hogyan lehet egyetlen művelettel ellenőrizni, hogy egy szám kettő hatványa-e?

10.3. Kifejezés-utasítás

A C nyelv legalapvetőbb utasítása a kifejezés-utasítás. Szintaxisa:

```
<kifejezés>;
```

A korábbi fejezetekben már találkoztunk a kifejezés-utasítással alapszinten. Példáink többnyire arra szorítkoztak, hogy egy értékadás révén egy változó felvette egy aritmetikai kifejezés értékét:

```
x = a + b;
```

Ebben a formában a kifejezés-utasítás szerepe magától értetődőnek tűnik: kiszámítunk egy értéket, majd eltároljuk azt. Mesterszinten azonban érdemes megállni ennél az egyszerű értelmezésnél, és megvizsgálni, mit jelent valójában az, hogy „egy kifejezés utasításként szerepel”.

10.3.1. A kifejezés-utasítás lényege: a hatás, nem az érték

Tömören fogalmazva a kifejezés-utasítás lényege a hatás, nem az érték. Formálisan minden kifejezésnek van értéke. A kifejezés-utasítás esetén azonban ez az érték eldobásra kerül. A fordító kiértékeli a kifejezést, majd az eredményt

nem használja fel semmire. Ez azt jelenti, hogy egy kifejezés-utasítás nem az értéke miatt érdekes, hanem kizárólag a mellékhatása miatt.

Például az `i++;` kifejezés-utasítás kapcsán az `i++` kifejezés értéke az inkrementálás előtti érték, de ez az érték itt irreleváns. Ami számít, az az, hogy a változó állapota megváltozik.

Ugyanez igaz a `x = y + 1;` kifejezés-utasítás esetén is: a kifejezés értéke maga a hozzárendelt érték, de az utasítás jelentése az, hogy `x` új értéket kap.

A lényeg: a kifejezés-utasítás olyan kifejezés, amelynek értéke érdektelen, de mellékhatása lényeges.

10.3.2. Milyen kifejezések alkalmasak utasításnak?

Elvileg bármely kifejezés írható utasításként. A gyakorlatban azonban csak azoknak van értelme, amelyek állapotváltozást okoznak. Ilyenek például:

- értékadások,
- inkrementálás és dekrementálás,
- függvényhívások mellékhatással,
- mutatón keresztüli módosítások.

Íme néhány példa:

```
scanf("%d", &x);     // függvényhívás mellékhatással
*p = 0;              // memória módosítása
a[i++] = x;          // index és tömbelem változik
```

Ezzel szemben az alábbi kifejezés-utasítás formailag helyes, de értelmetlen:

```
a + b;
```

Ez a kifejezés kiszámít egy értéket, de semmilyen mellékhatása nincs. A program futása szempontjából teljesen hatástalan. Egy ilyen sor rendszerint programozási hiba, vagy félkész kód nyoma.

Következtetés: egy kifejezés-utasítást nem az alapján kell értelmezni, hogy mit számol ki, hanem az alapján, hogy mit változtat meg.

10.3.3. Függvényhívás mint kifejezés-utasítás

A kifejezés-utasítás egyik leggyakoribb formája a függvényhívás:

```
printf("Hello!\n");
```

Formailag a `printf` egy értéket ad vissza (a kiírt karakterek számát), de ez az érték itt nem érdekes. A kifejezés-utasítás jelentése kizárólag a mellékhatás: a kimenet módosítása. Ugyanez igaz sok más függvényre is, például a `scanf`-re.

Az érem másik oldala, hogy nem minden függvényhívás „utasítás jellegű". Egy pusztán számoló függvény hívása utasításként gyanús:

```
sqrt(x);   // értelmetlen, ha az eredményt nem használjuk fel
```

Ez a sor nem változtatja meg a program állapotát, így hatástalan. Ha mégis ilyen kódot látunk, az vagy hiba, vagy félreértés.

10.3.4. Kifejezés-utasítás és vezérlés

A kifejezés-utasítások gyakran jelennek meg vezérlési szerkezetek törzsében:

```
while (i < n) {
    i++;
}
```

Itt a ciklus törzse egyetlen kifejezés-utasításból áll. A ciklus működése teljes egészében ezen az egy mellékhatáson alapul. Ez jól mutatja, hogy a C-ben a vezérlés és az állapotváltozás nem válik el élesen: a vezérlés gyakran kifejezés-utasítások láncolatán keresztül valósul meg.

10.3.5. Túl tömör kifejezés-utasítás veszélye

A C nyelv lehetővé teszi, hogy egyetlen kifejezés-utasítás több mellékhatást is hordozzon.

Például a

```
a[i++] = b[j--];
```

kifejezés-utasítás formailag helyes, de olvasásakor a programozónak egyszerre kell követnie, hogy:

- melyik változó mikor változik,
- milyen sorrendben történnek a mellékhatások,
- milyen érték kerül hova.

Mesterszinten ez nem technikai, hanem stilisztikai kérdés: az ilyen kifejezés-utasítás akkor elfogadható, ha a jelentése azonnal átlátható. Ha nem, akkor célszerűbb több egyszerűbb kifejezés-utasításra bontani.

10.3.6. Üres és összetett utasítás – szintaktikai kényszer, szemantikai szabadság

A C nyelv vezérlési szerkezeteinek szintaxisa sok esetben pontosan egyetlen utasítást enged meg. Ez a megkötés független attól, hogy a program logikája szerint hány műveletet kell végrehajtani, vagy éppen szükség van-e bármilyen műveletre.

Ha a vezérlés helyén több műveletet kell végrehajtani, azokat összetett utasítással vonjuk össze egyetlen egységgé:

```
if (x > 0) {
    y = x;
    z = x * x;
}
```

A kapcsos zárójel itt nem külön vezérlési szerkezet, hanem azt fejezi ki, hogy a két kifejezés-utasítás egy utasításként viselkedik a szintaxis szempontjából.

Előfordulhat viszont az is, hogy a szintaxis által megkövetelt utasítás helyén nincs tényleges végrehajtandó művelet. Ilyenkor alkalmazható az üres utasítás, amely formailag kielégíti a szintaktikai követelményt, de nem jár mellékhatással:

```
while (*p++);
```

Ebben az esetben a ciklus feltétele tartalmazza az összes lényeges műveletet, a ciklustörzsnek nincs önálló szerepe.

Az összetett és az üres utasítás tehát ugyanarra a szintaktikai megkötésre ad két ellentétes választ: az egyik lehetővé teszi több művelet egyetlen utasításként való kezelését, a másik pedig azt, hogy akkor is eleget tegyünk a nyelv formai követelményeinek, amikor nincs végrehajtandó művelet.

[i] Megjegyzés. Bár a C szintaxisa megengedi, hogy egy vezérlési szerkezet törzse egyetlen, kapcsos zárójelek nélküli utasítás legyen, a gyakorlatban gyakran ajánlott az összetett utasítás használata akkor is, ha csak egy kifejezés-utasítást tartalmaz. A kapcsos zárójelek ilyenkor nem technikai szükségszerűséget, hanem szándékot fejeznek ki: egyértelművé teszik, mely utasítás tartozik a vezérléshez, és védelmet nyújtanak a későbbi módosítások során fellépő rejtett hibák ellen.

▶ Összegzés

A kifejezés-utasítás a C nyelv legkisebb végrehajtható egysége. Jelentése nem az értékében, hanem a mellékhatásában rejlik. Aki mesterszinten használja a C-t, az nem pusztán „leírható” kifejezésekben gondolkodik, hanem tudatosan mérlegeli, hogy egy adott állapotváltozás kifejezésként elég-e, vagy inkább explicit utasításszerkezetet igényel.



AI-csepp
Kérdezd meg az AI-t!
• Mit jelent az, hogy egy kifejezésnek „nem definiált viselkedése” van?
• Adj példát olyan kifejezésre, ahol a mellékhatások sorrendje nem meghatározott!

10.4. Értékvezérelt elágazás: a `switch` utasítás

A vezérlési szerkezetek közül az elágazással már korábban, az 5. fejezetben megismerkedtünk. Az `if` és az `if-else` szerkezetek általános döntési mechanizmust adnak a kezünkbe: egy logikai kifejezés igaz vagy hamis értéke alapján választanak végrehajtási utat. Ez a forma rugalmas, és szinte minden feltételes helyzet leírható vele.

Vannak azonban olyan esetek, amikor a döntés nem összetett logikai feltételeken alapul, hanem azon, hogy egy kifejezés értéke melyik konkrét alternatívával egyezik meg. Tipikus példa, amikor egy változó értéke több lehetséges konstans egyikét veheti fel, és ezekhez eltérő műveletek tartoznak. Ilyenkor az egymásba ágyazott vagy hosszú `if-else` `if-else` lánc nemcsak nehezen olvashatóvá válik, hanem a döntési logika szerkezete is elmosódik.

A C nyelv erre a problémára kínál külön vezérlési szerkezetet: a `switch` utasítást. A `switch` nem általánosabb, mint az `if`, hanem szűkebb, de célzottabb eszköz. Akkor használható, amikor egyetlen kifejezés értékét hasonlítjuk össze diszkrét, konstans értékek halmazával. Cserébe a döntési struktúra áttekinthetőbbé válik, és a program logikája közelebb kerül ahhoz, ahogyan a problémát gondolatban megfogalmazzuk: „az értéktől függően ezt vagy azt tesszük”.

10.4.1. A `switch` szerkezet

A switch utasítás egy kifejezés értékét vizsgálja, és annak eredményét hasonlítja össze előre megadott konstans értékekkel. Általános alakja:

```
switch (kifejezés) {
    case konstans1:
        /* utasítások */
        break;
    case konstans2:
        /* utasítások */
        break;
    default:
        /* utasítások */
}
```

A `switch` működésének kulcsa az, hogy nem feltételkifejezéseket, hanem értékegyezést vizsgál. A kifejezés egyszer kerül kiértékelésre, majd az eredmény alapján a vezérlés arra a `case` címkére ugrik, amelynek konstans értéke megegyezik vele.

A `default` címke akkor kerül végrehajtásra, ha a vizsgált kifejezés értéke egyik `case` konstanssal sem egyezik meg. Szerepe általában nem egy „utolsó lehetőség” biztosítása, hanem egy hibakezelési vagy védelmi ág megvalósítása az előre nem várt esetek kezelésére.

Az alábbi példában egy 1 és 5 közötti egész számhoz tartozó szöveges minősítést írunk ki. A döntési logikát először `switch` szerkezettel, majd azzal ekvivalens `if-else` lánccal valósítjuk meg.

```
/* Megvalósítás switch-el */
switch (ertekeles) {
    case 1:
        printf("Elegtelen\n");
        break;
    case 2:
        printf("Elegseges\n");
        break;
    case 3:
        printf("Kozepes\n");
        break;
    case 4:
        printf("Jo\n");
        break;
    case 5:
        printf("Kivalo\n");
        break;
    default:
        printf("Ervenytelen ertekeles\n");
        break;
}

/* Megvalósítás if-else lánccal */
if (ertekeles == 1) {
    printf("Elegtelen\n");
} else if (ertekeles == 2) {
    printf("Elegseges\n");
} else if (ertekeles == 3) {
    printf("Kozepes\n");
} else if (ertekeles == 4) {
    printf("Jo\n");
} else if (ertekeles == 5) {
    printf("Kivalo\n");
```

```
} else {
    printf("Ervenytelen ertekeles\n");
}
```

Ebben a példában a `switch` előnye elsősorban abban rejlik, hogy a döntési logika áttekinthetőbbé válik. Bár az `if-else` szerkezetek általánosabb eszközt adnak a feltételes vezérléshez, ebben az esetben kevésbé fejezik ki azt, hogy egyetlen kifejezés értékét hasonlítjuk össze több lehetséges konstanssal.

10.4.2. A „`switch-break`” és a fall-through jelenség

Fontos hangsúlyozni: a `case` címkék nem blokkok, hanem ugrási célpontok. A vezérlés a megfelelő címkétől kezdve folytatólagosan hajtja végre az utasításokat, egészen addig, amíg el nem ér egy `break` utasítást, vagy a `switch` végére nem jut. Más szóval, a `switch` működésében a `break` határjel: azt fejezi ki, meddig tart egy adott döntési ág hatóköre. Enélkül a végrehajtás átcsúszik a következő `case`-be is. Ezt a jelenséget nevezzük fall-through működésnek.

Íme egy példa arra, miként használható ki tudatosan a fall-through jelenség. Tegyük fel, hogy egy rendszerben a felhasználók jogosultsági szintekkel rendelkeznek, és egy magasabb szint automatikusan magában foglalja az alacsonyabb szintekhez tartozó jogosultságokat is. Ezt a hierarchikus logikát a switch szerkezet természetes módon képes kifejezni.

Az alábbi példában a `jogosultsag` egy egész típusú változó, míg az `ADMIN`, `SZERKESZTO` és `OLVASO` szimbolikus egész konstansok:

```
switch (jogosultsag) {
    case ADMIN:
        torol_felhasznalo();
        /* szándékos fall-through */
    case SZERKESZTO:
        modosithat_tartalom();
        /* szándékos fall-through */
    case OLVASO:
        megtekinthet_tartalom();
        break;
    default:
        hiba("Ismeretlen jogosultsag");
        break;
}
```

Ebben a szerkezetben egy `ADMIN` jogosultságú felhasználó minden alacsonyabb szinthez tartozó műveletet is végrehajthat, míg egy `SZERKESZTO` nem törölhet

felhasználót, de módosíthat és megtekinthet tartalmat. Az `OLVASO` jogosultság kizárólag megtekintést tesz lehetővé.

A `break` utasítások hiánya az egyes `case` ágak között nem hiba, hanem tudatos tervezési döntés: a vezérlés szándékosan „átcsúszik” a következő ágakba, így fejezve ki a jogosultságok egymásra épülését.

Ugyanez a logika természetesen `if-else` lánccal is megvalósítható:

```
if (jogosultsag == ADMIN) {
    torol_felhasznalo();
    modosithat_tartalom();
    megtekinthet_tartalom();
} else if (jogosultsag == SZERKESZTO) {
    modosithat_tartalom();
    megtekinthet_tartalom();
} else if (jogosultsag == OLVASO) {
    megtekinthet_tartalom();
}
```

Ez a megoldás azonban több szempontból is hátrányos. Az egyes műveletek ismétlődnek, ami redundanciához vezet, így a kód nehezebben karbantartható: egy közös művelet módosítása több helyen igényel beavatkozást. Emellett a jogosultságok hierarchikus viszonya sem jelenik meg közvetlenül a szerkezetben, csupán implicit módon következtethető ki a kódból. Ezzel szemben a `switch` és a tudatos fall-through használata nemcsak tömörebb, hanem a döntési logika szerkezetét is világosabban fejezi ki.

Összefoglalva: a fall-through használata akkor indokolt, ha az egyes `case` ágak nem független alternatívák, hanem egymásra épülő műveletsorokat képviselnek.

10.4.3. A `switch` korlátai

A `switch` tudatosan korlátozott eszköz:

- a vizsgált kifejezés típusa egész típus kell legyen (`int`, `char`),
- a `case` értékek fordítási idejű konstansok,
- nem használható tartományokra, relációkra vagy összetett feltételekre.

Ezek a korlátok azonban nem gyengeségek, hanem a `switch` szerepét definiálják: diszkrét állapotok közötti választás.



AI-csepp

Kérdezd meg az AI-t!

- Miért kell a `case` értékeknek fordítási idejű konstansoknak lenniük?

10.5. Az ismétlés mint vezérlési modell

A 6. és 7. fejezetekben megismertük a C nyelv ciklusutasításait (`while`, `do-while`, `for`), valamint a `break` és `continue` használatát. Ott az alapmintákra koncentráltunk: számlálás, feltételes ismétlés, adatsorok bejárása stb.

Ebben az alfejezetben a ciklusokat magasabb nézőpontból vizsgáljuk. Nem új szerkezetet tanulunk, hanem azt elemezzük, milyen vezérlési modellt fejeznek ki, és miként tükrözi a ciklus formája a feladat természetét. Két alapvető szemléletet különböztethetünk meg: feltételvezérelt ismétlés, eseményvezérelt ismétlés.

10.5.1. Feltételvezérelt ismétlés

Feltételvezérelt ciklus esetén az ismétlés addig tart, amíg egy előre megfogalmazott logikai feltétel igaz. A ciklus hossza tehát egy állapot logikai értékétől függ.

Például az állományvégig történő beolvasás természetes modellje: amíg sikerül adatot beolvasni, addig dolgozzuk fel.

```
// összegszámítás állományvégig olvasott egész számokon
int x;
int osszeg = 0;

while (scanf("%d", &x) == 1) {
    osszeg += x;
}
```

A ciklus feltétele egyértelműen megfogalmazható, és a megszűnése a feltétel hamissá válásából következik.

10.5.2. Eseményvezérelt ismétlés

Eseményvezérelt ciklus esetén az ismétlés nem egyetlen fejlécbeli feltételhez kötött. A ciklus elvileg nyitott, és egy belső esemény – például végjel vagy felhasználói döntés – szakítja meg.

Az eseményvezérelt ciklus természetes eszköze a végtelen ciklus:

```
// végtelen for: hiányzik a feltétel
for (;;) {
    /* ... */
}
// végtelen while: a feltétel igaz értékű konstans
while (1) {
    /* ... */
}
```

A ciklusból egy vagy több `break` utasítás vezet ki.

Az alábbi példában egész számokat olvasunk be adott végjelig, egészen pontosan 0 végjelig, és ezek összegét számoljuk ki.

```
int x;
const int VEGJEL = 0;
int osszeg = 0;

for (;;) {
    if (scanf("%d", &x) != 1)
        break;          /* allomanyveg */

    if (x == VEGJEL)
        break;          /* vegjel */

    osszeg += x;
}
```

Itt a ciklust jól definiált események szakítják meg: az állományvég, illetve a logikai végjel (0). A megszakítás nem a fejlécben, hanem a ciklustörzsben jelenik meg. Ez a megközelítés különösen természetes, ha több, eltérő jellegű megállási feltétellel van dolgunk, hiszen ezek egyetlen ciklusfeltételbe sűrítése nehézkes vagy kevésbé áttekinthető lehet.

A ciklus elvileg végtelen, de világosan meghatározott kilépési pontokkal rendelkezik. A végtelen ciklus önmagában nem hiba; problémává akkor válik, ha nincs egyértelműen definiált megszakítási mechanizmusa.

10.5.3. Menüvezérelt program: minikalkulátor két szemléletben

A menüvezérelt program jó példa arra, hogy ugyanaz a feladat mindkét vezérlési modellben megfogalmazható.

Az alábbi megvalósításban az eseményvezérelt modellt követtük, végtelen ciklust alkalmazva. A példában feltételeztük, hogy a beolvasások sikeresek lesznek, illetve nem fordul elő nullával való osztás.

A `printf` hívásokban a `%g` formátumkaraktert használtuk, amely a lebegőpontos számokat „általános” alakban jeleníti meg. Erről a formátumról korábban nem volt szó. Jó gyakorlat lehet megkérdezni egy AI-eszköztől: (i) miben különbözik a `%f`, `%e` és `%g`; (ii) milyen szabály alapján választ a `%g` a két forma között; (iii) hogyan értelmezendő a precízió megadása (`%.10g`).

```
for (;;) {
    printf("\n1 - Osszeadas\n");
```

```
        printf("2 - Kivonas\n");
        printf("3 - Szorzas\n");
        printf("4 - Osztas\n");
        printf("0 - Kilepes\n");
        printf("Valasztas: ");

        int valasztas;
        scanf("%d", &valasztas);

        if (valasztas == 0) { break; }

        double a, b;
        printf("Add meg a ket szamot: ");
        scanf("%lf %lf", &a, &b);

        switch (valasztas) {
            case 1:
                printf("Eredmeny: %.10g\n", a + b);
                break;
            case 2:
                printf("Eredmeny: %.10g\n", a - b);
                break;
            case 3:
                printf("Eredmeny: %.10g\n", a * b);
                break;
            case 4:
                printf("Eredmeny: %.10g\n", a / b);
                break;
            default:
                printf("Ervenytelen valasztas.\n");
        }
    }
```

Ebben a modellben a ciklus a program működési kerete, a kilépés pedig belső eseményhez kötött (a 0 választás). A `break` utasítás explicit megszakítási pontot jelöl.

Ugyanez a működés feltételvezérelt formában is megfogalmazható, például `do-while` ciklus segítségével; ennek megvalósítását gyakorlatként az olvasóra bízzuk.

Az eseményvezérelt modellben a ciklus elvileg végtelen, és a kilépés explicit megszakítás eredménye. A feltételvezérelt modellben ezzel szemben a ciklus megszűnése a feltétel hamissá válásából következik. Mindkét forma korrekt és strukturált; a választás a hangsúlytól és a feladat természetétől függ.

10.5.4. Többszörös egymásba ágyazás

Az egymásba ágyazott ciklusokkal már találkoztunk, például a kétdimenziós tömbök bejárásánál: a külső ciklus a sorokat járta be, a belső pedig az aktuális soron lépegetett oszlopról oszlopra. Most azt a helyzetet vizsgáljuk, amikor a belső (vagy még belsőbb) ciklus önálló részfeladatot, illetve rész-részfeladatot valósít meg.

A prímvizsgálat alapötletével már találkoztunk a 7.5. alfejezetben. Most azt emeljük ki, hogy a prímteszt a külső feldolgozáson belül önálló részfeladatként jelenik meg.

```
for (i = 0; i < n; i++) {
    int szam = tomb[i];
    int prim = 1;

    if (szam < 2)
        prim = 0;
    else {
        for (j = 2; j * j <= szam; j++) {
            if (szam % j == 0) {
                prim = 0;
                break;
            }
        }
    }

    if (prim)
        printf("%d prim\n", szam);
}
```

A szerkezet hierarchikus: a külső ciklus az elemek feldolgozását végzi azáltal, hogy elemről elemre halad, míg a belső ciklus az aktuális elem prímvizsgálatát végzi el.

A `break` utasítással kapcsolatban fontos szabály, hogy mindig csak azt a ciklust szakítja meg, amelynek törzsében közvetlenül szerepel – jelen esetben a prímtesztet végző belső ciklust.

A belső ciklusok gyakran önálló jelentéssel bíró részfeladatokat hordoznak. Ezt a példában azzal is hangsúlyoztuk, hogy a prímvizsgálatot a tömb aktuális elemének másolatán (`szam = tomb[i]`) végeztük el, így a belső rész logikailag elkülönül a külső feldolgozástól. Ahogy ezek a részfeladatok összetettebbé válnak, a ciklustörzs egyre hosszabb és nehezebben áttekinthető lesz.

Ezen a ponton természetes igény jelenik meg a program szerkezetének tagolására. A következő fejezetben azt vizsgáljuk meg, hogyan választhatók le ezek a

részfeladatok függvények segítségével, és miként emelhető a program felépítése magasabb absztrakciós szintre.



AI-csepp

Kérdezd meg az AI-t!

- Hogyan fejezi ki a ciklus formája a probléma természetét?
- Hogyan függ össze a mellékhatás és a vezérlés a C-ben?
- Miért mondható, hogy a C „nem véd meg önmagadtól”?

►► Zárógondolat

A C nyelvben a vezérlés nem különálló mechanizmus, hanem a kifejezések természetes következménye. A program nem utasítások laza sorozata, hanem egymásba ágyazott kifejezések és állapotváltozások rendszere. A döntések, az ismétlések és az elágazások mind ugyanarra az alapra épülnek: egy kifejezés értékére és annak mellékhatásaira.

Ez a szemlélet egyszerre ad erőt és ró felelősséget a programozóra. A C nagy szabadságot biztosít: megengedi a tömörséget, az egymásba ágyazott mellékhatásokat, a vezérlés és az állapot szoros összefonódását. Ugyanakkor nem véd meg a túlzott tömörségtől, a nem definiált viselkedéstől vagy az átláthatatlan szerkezetektől. A helyes és világos program tehát nem a nyelv kényszerének, hanem a programozó tudatos döntéseinek eredménye.

Aki mesterszinten használja a C-t, az nem pusztán ismeri az operátorokat és vezérlési szerkezeteket, hanem érti azok mögöttes logikáját. Felismeri, mikor természetes egy feltételvezérelt ciklus, mikor indokolt az eseményvezérelt keret, mikor fejez ki többet egy `switch`, és mikor válik veszélyessé egy túl sűrített kifejezés.

Ezen a ponton a C már nem eszköz, hanem gondolkodási forma: a program szerkezete a probléma szerkezetét tükrözi. Amikor ez az összhang létrejön, akkor válik igazán láthatóvá a C nyelv valódi arca.

11. Függvények: részfeladatok saját kézben

Az eddigi programjainkban többször találkoztunk olyan helyzettel, amikor egy részfeladat elvégzését nem közvetlenül mi írtuk le utasításról utasításra, hanem „leosztottuk” egy már létező függvénynek. Amikor adatot olvastunk be, a `scanf` függvényt hívtuk segítségül. Amikor eredményt írtunk ki, a `printf` végezte el helyettünk a formázott megjelenítés feladatát. Amikor fájlból szerettünk volna adatokat olvasni, a `freopen` segítségével irányítottuk át a bemenetet. A másodfokú egyenlet megoldásánál pedig a négyzetgyökvonást a `sqrt` függvényre bíztuk.

Ezekben az esetekben közös, hogy a programunk egy adott pontján szükségünk volt egy jól körülhatárolható részfeladat elvégzésére, és ehhez már rendelkezésünkre állt egy megfelelő könyvtári függvény. A függvényhívás ilyenkor olyan, mintha a program egy rövid utasítással megbízna egy másik, előre megírt programrészt: „olvasd be ezt az adatot”, „írd ki ezt az értéket”, „számítsd ki ennek a számnak a négyzetgyökét”.

De mi történik akkor, ha egy részfeladatra nincs kész könyvtári függvény? Mi van akkor, például, ha azt szeretnénk eldönteni, hogy egy szám prím-e? Vagy ki szeretnénk számítani egy szám számjegyeinek összegét? Vagy többször is szükségünk van arra, hogy két szám közül meghatározzuk a nagyobbat? Ilyenkor természetesen megírhatjuk a szükséges utasításokat minden alkalommal újra és újra. Ez azonban hamar ismétlődéshez, hosszú és nehezen áttekinthető programokhoz vezetne.

A C nyelv lehetőséget ad arra, hogy saját függvényeket hozzunk létre. Ez azt jelenti, hogy egy gyakran használt vagy önálló jelentéssel bíró részfeladatot külön programrészbe szervezünk, nevet adunk neki, majd később ugyanúgy meghívhatjuk, mint a könyvtári függvényeket. A különbség csupán annyi, hogy ezúttal nem a C szabványos könyvtára, hanem mi magunk írjuk meg a függvény működését.



Ebben a fejezetben azt vizsgáljuk meg, hogyan lehet egy programot kisebb, jól érthető részekre bontani. Megtanuljuk, hogyan definiálhatunk saját függvényeket, hogyan adhatunk át nekik adatokat paraméterek segítségével, és hogyan kaphatunk vissza tőlük eredményt visszatérési érték formájában. Közben azt is látni fogjuk, hogy a függvények nem pusztán kényelmi eszközök: a program szerkezetének, olvashatóságának és újrafelhasználhatóságának alapvető építőelemei.

11.1. Függvénydefiníció, függvénydeklaráció és függvényhívás

A könyvtári függvények esetében eddig többnyire csak a függvényhívással találkoztunk: megadtuk a függvény nevét, zárójelben átadtuk neki a szükséges adatokat, majd felhasználtuk az általa elvégzett művelet eredményét vagy hatását.

Most először a másik oldalra is átlépünk: nemcsak meghívunk egy függvényt, hanem mi magunk hozunk létre egyet. Első példaként írjunk egy olyan függvényt, amely két egész szám közül visszatéríti a nagyobbat.

Tegyük fel, hogy szeretnénk két beolvasott egész szám közül kiírni a nagyobbat. Milyen kényelmes lenne, ha egyszerűen így írhatnánk:

```
printf("A nagyobb szam: %i\n", max(x, y));
```

Ez a sor úgy néz ki, mintha a `max` egy könyvtári függvény lenne, amely két szám közül visszatéríti a nagyobbat. Az alap C nyelvben azonban nincs általános `max` függvény egész számokra. Mit tehetünk, ha mégis szeretnénk egy ilyen függvényt használni, mégpedig ugyanúgy, ahogyan a könyvtári függvényeket már használni tudjuk?

A megoldás az, hogy megírjuk a saját `max` függvényünket. Ennek tartalmaznia kell a maximum meghatározásának forgatókönyvét: össze kell hasonlítania a két számot, majd vissza kell adnia közülük a nagyobbat.

```
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

Ezt a programrészt egy saját függvény, jelen esetben a `max` függvény *definíciójának* nevezzük. A függvénydefiníció két fő részből áll: a függvény fejrészéből és a függvény törzséből.

A fejrész:

```
int max(int a, int b)
```

</> Szintaxis: a függvény fejrészének általános alakja

```
<visszatérési_típus> <függvénynév>(<paraméterlista>)
```

A `max` függvény esetében ez azt jelenti, hogy a függvény neve max, két `int` típusú adatot vár, és egy `int` típusú eredményt térít vissza. Az `a` és `b` paraméterek olyan helyi változók, amelyek a függvény meghívásakor kapnak értéket.

A függvény törzse kapcsos zárójelek között található. Ebben írjuk le, hogyan kell a részfeladatot elvégezni. A `return` utasítás visszaadja a függvény eredményét annak a programrésznek, amely a függvényt meghívta.

Hova írjuk a saját függvényünk definícióját? Az egyik lehetőség az, hogy a `main` függvény elé helyezzük.

```
#include <stdio.h>

int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int x, y;

    scanf("%i %i", &x, &y);

    printf("A nagyobb szam: %i\n", max(x, y));

    return 0;
}
```

Ebben a változatban mire a fordító a `main` függvényben találkozik a `max(x, y)` hívással, már ismeri a `max` függvényt: tudja a nevét, a paramétereit, a visszatérési típusát és a működését is. Ennek alapján ellenőrizni tudja, hogy helyesen hívtuk-e meg.

Nagyobb programokban azonban áttekinthetőbb lehet, ha előbb a `main` függvényt látjuk, és csak utána következnek a részfeladatokat megvalósító függvények definíciói. Ilyenkor a `main` elé nem a teljes függvénydefiníciót írjuk, hanem csak a függvény *deklarációját*, más néven *függvényprototípusát*.

A deklaráció a függvény „ígérete”: még nem mondja meg, hogyan működik a függvény, de közli a fordítóval a függvény használatához szükséges kulcsinformációkat. Pontosan annyit tartalmaz, amennyi a függvényhívás helyességének ellenőrzéséhez szükséges.

```
int max(int, int);
```

Ez azt közli a fordítóval, hogy létezik egy `max` nevű függvény, amely két `int` típusú adatot kap, és `int` típusú értékkel tér vissza. A paraméterek nevei a deklarációban elhagyhatók, hiszen a fordító számára itt elsősorban a típusok fontosak.

A függvény tényleges működését majd később, a `main` után adjuk meg, a függvénydefiníció révén.

A program így is megírható:

```
#include <stdio.h>

int max(int, int);   // függvénydeklaráció

int main() {
    int x, y;

    scanf("%i %i", &x, &y);

    printf("A nagyobb szam: %i\n", max(x, y));

    return 0;
}

int max(int a, int b) {  // függvénydefiníció
    if (a > b) {
        return a;
    } else {
        return b;
    }
}
```

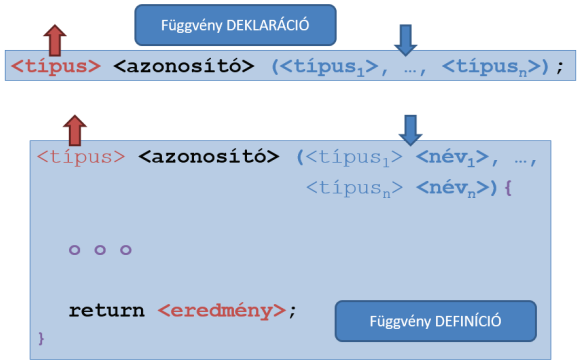
A `main` függvényben szereplő `max(x, y)` kifejezés a függvény *hívása*. A hívás azt jelenti, hogy a program végrehajtatja a `max` függvényt a konkrétan megadott adatokkal. Ebben az esetben az `x` és `y` értéke adódik át a függvénynek. A `max` függvény elvégzi az összehasonlítást, majd a `return` utasítással visszatéríti a nagyobb értéket. Ez az érték kerül be a `printf` függvényhívásba.

A 11.1. táblázat összefoglalja a függvényekhez kapcsolódó alapfogalmakat és szerepüket, a 11.1. ábra pedig egy függvény deklarációját és definícióját mutatja be egymás mellett, valamint szemlélteti a hívó és a hívott függvény közötti kétirányú kapcsolatot.

11.1. táblázat. *A függvényekhez kapcsolódó alapfogalmak és szerepük*

Fogalom	Szerepe
függvénydeklaráció	jelzi a fordítónak, hogy létezik a függvény, és megadja a híváshoz szükséges típusinformációkat
függvénydefiníció	megadja a függvény tényleges működését
függvényhívás	végrehajtatja a függvényt egy konkrét helyzetben
hívó függvény	az a függvény, amelyből a másik függvényt meghívjuk
hívott függvény	az a függvény, amelynek végrehajtását a hívás elindítja

A fenti példában a `main` a hívó függvény, a `max` pedig a hívott függvény. Amikor azonban a `main` a `printf` függvényt hívja meg, akkor ebben a kapcsolatban a `main` a hívó függvény, a `printf` pedig a hívott függvény.



11.1. ábra. *A függvény deklarációja és definíciója egymás mellett, valamint a hívó és a hívott függvény közötti kétirányú kapcsolat. A kék nyilak azt jelzik, hogy a hívott függvény a paraméterlistán keresztül adatokat kap a hívó függvénytől, míg a piros nyilak a visszatérési érték továbbítását szemléltetik.*

[i] Megjegyzés. A könyvtári függvények esetében is ugyanez a mintázat érvényesül. A 3. fejezetben azt mondtuk, hogy a bemeneti és kimeneti műveleteket lebonyolító könyvtári függvények – például a `scanf`, a `printf` vagy a `freopen` – használata feltételezi a `#include <stdio.h>` programsor jelenlétét. Most már pontosabban is értjük, miért van erre szükség: az `stdio.h` fejlécállomány tartalmazza ezeknek a függvényeknek a deklarációit. Ezek alapján a fordító ellenőrizni tudja, hogy helyesen hívjuk-e meg őket. A függvények tényleges definíciói a könyvtárban találhatók; ezért nevezzük őket könyvtári függvényeknek.

[i] Megjegyzés. A `main` függvény különleges szerepet tölt be. A program futtatásakor ez hívódik meg elsőként, nem pedig egy általunk írt másik függvényből. A `main`

függvény esetében általában nem használunk paramétereket, ezért eddigi programjainkban is így írtuk fel: `int main()`. A következő mesterfejezetben azonban látni fogjuk, hogy a `main` függvénynek létezik paraméteres alakja is, amelynek segítségével a program indításakor megadott adatok is átvehetők. A `return 0;` utasítás pedig azt jelzi a futtatókörnyezetnek, hogy a program rendben, hiba nélkül fejeződött be.

[i] Megjegyzés. Nem minden függvény feladata az, hogy értéket számítson ki és térítsen vissza. Vannak olyan függvények, amelyek egyszerűen elvégeznek egy műveletet: például kiírnak valamit, módosítanak egy adatot, vagy lebonyolítanak egy részfeladatot. Ilyenkor a függvény visszatérési típusa `void`.

```
void udvozol() {
    printf("Udvozollek!\n");
}
```

A `void` szó ebben az esetben azt jelenti, hogy a függvény nem térít vissza értéket a hívás helyére. Más programozási nyelvek szóhasználatával az ilyen programrészt eljárásnak is nevezhetnénk, a C nyelvben azonban ezt is függvényként kezeljük. Egy `void` függvényben szerepelhet üres `return` utasítás, amely az adott ponton azonnal befejezi a függvény végrehajtását.

Gyakorló feladatok

A függvények egyik legfontosabb szerepe az, hogy egy jól körülhatárolható részfeladatot kiemeljünk a program fő gondolatmenetéből, nevet adjunk neki, majd később újra és újra meghívhassuk. Ezért a saját függvények gyakorlásához érdemes olyan részfeladatokból kiindulni, amelyekkel korábban már találkoztunk.

Az alábbi feladatokban nem az algoritmusok újak, hanem a szervezési mód. A cél az, hogy a korábban megismert számításokat és ellenőrzéseket saját függvényként írjuk meg. Így a `main` függvény áttekinthetőbbé válik: nem minden részletet tartalmaz, hanem a részfeladatokat beszédes nevű függvényekre bízza.

Minden feladatnál megadtuk a megírandó függvény deklarációját. A feladat tehát elsősorban a függvénydefiníció elkészítése: meg kell írnod a függvény törzsét, vagyis azt az utasítássorozatot, amely a részfeladatot megoldja. Minden megoldáshoz írj egy `main` függvényt is, amelyben beolvasod a szükséges adatokat, meghívod a szóban forgó függvényt, majd kiírod az eredményt.

- Írj függvényt annak eldöntésére, hogy egy egész szám páros-e!

```
int paros_e(int n);
```

- Írj függvényt annak eldöntésére, hogy egy pozitív egész szám prím-e! (7.5. alfejezet)

```
int prim_e(int n);
```

- Írj függvényt két pozitív egész szám legnagyobb közös osztójának meghatározására! (7.3. alfejezet)

```
int lnko(int a, int b);
```

- Írj függvényt két pozitív egész szám legkisebb közös többszörösének meghatározására! (7.3. alfejezet)

```
int lkkt(int a, int b);
```

- Írj függvényt egy pozitív egész szám számjegyeinek összegére! (7.1.1. alfejezet)

```
int szamjegyosszeg(int n);
```

- Írj függvényt annak eldöntésére, hogy egy pozitív egész szám Harshad-szám-e! (7.1.1. alfejezet)

```
int harshad_e(int n);
```

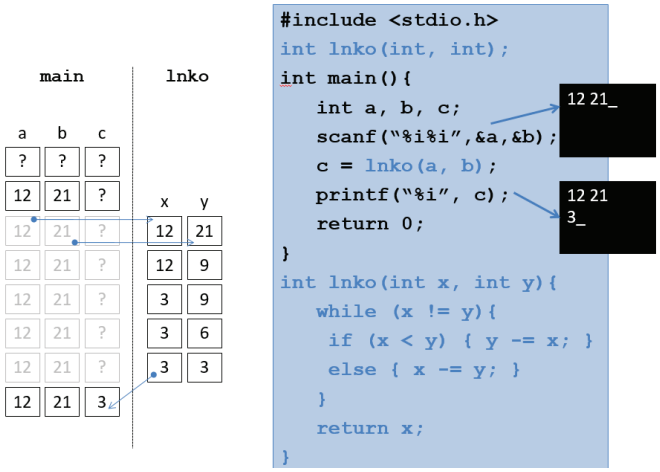
11.2. Paraméterátadás

A `max` függvény példája arra is alkalmas, hogy megértsük a paraméterátadás alapműködését. Amikor a `main` függvényben ezt írjuk, hogy `max(x, y)`, akkor az `x` és `y` a hívás *argumentumai*. Ezek azok a konkrét értékek, amelyeket a függvénynek átadunk. A függvény definíciójában szereplő `a` és `b` viszont *paraméterek*: `int max(int a, int b)`.

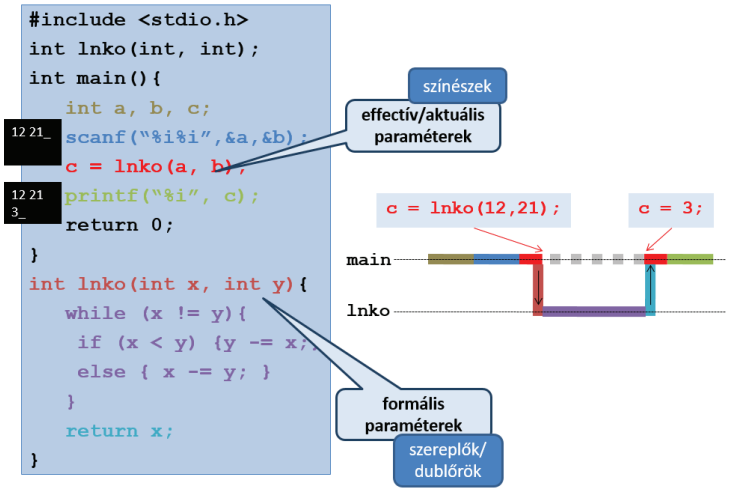
A hívás pillanatában az `x` *értéke* bemásolódik az a paraméterbe, az `y` értéke pedig a `b` paraméterbe. A függvény tehát nem az eredeti `x` és `y` változókkal dolgozik, hanem azok értékének másolatával. Ezt nevezzük *érték szerinti paraméterátadásnak*.

A 11.2. és 11.3. ábra a hívó és a hívott függvény közötti, érték szerinti, kétirányú kommunikációt szemlélteti.

[i] Megjegyzés. A szakirodalomban a paraméter és az argumentum elnevezések mellett más szóhasználattal is találkozhatunk. A függvény definíciójában szereplő paramétereket gyakran *formális paramétereknek* nevezzük, míg a híváskor átadott argumentumokat *aktuális paramétereknek* vagy *effektív paramétereknek* is szokás nevezni.



11.2. ábra. *Érték szerinti, kétirányú kommunikáció a hívó (`main`) és a hívott (`lnko`) függvény között. Az `a`, `b` és `c` a függvényhívásban szereplő argumentumok (aktuális paraméterek), míg az `x` és `y` a hívott függvény paraméterei (formális paraméterek).*



11.3. ábra. *A függvényhívás és a függvényből való visszatérés szemléltetése. Ha a függvényt egyfajta forgatókönyvként értelmezzük, akkor a formális paraméterek a szerepeknek, az aktuális paraméterek pedig az ezeket alakító színészeknek felelnek meg. A függvény meghívásakor a formális paraméterek az aktuális paraméterek értékei (dublőrjei) révén mintegy „életre kelnek”.*

A `max` függvényben az a és `b` saját, helyi változók. Ha a függvényen belül megváltoztatnánk az `a` vagy `b` értékét, az nem módosítaná a `main` függvényben található `x` és `y` változókat. A paraméterek a függvény lokális változói: a híváskor jönnek létre, kapnak kezdőértéket, majd a függvény befejezésekor megszűnnek.

Ez a működés a `max` esetében teljesen megfelelő, hiszen a függvénynek nem kell megváltoztatnia az eredeti változókat. Csupán meg kell vizsgálnia két értéket, és vissza kell adnia közülük a nagyobbat. Más a helyzet akkor, ha egy függvény feladata éppen az lenne, hogy módosítsa az eredeti változókat. Tegyük fel, hogy szeretnénk írni egy függvényt, amely felcseréli két egész változó értékét. Első próbálkozásként talán így írnánk meg:

```
void csere(int a, int b) {
    int seged = a;
    a = b;
    b = seged;
}
```

Ahogy már utaltunk rá, a `void` visszatérési típus azt jelenti, hogy a függvény nem térít vissza értéket, csak elvégez egy feladatot. A csere látszólag megtörténik: az `a` és `b` paraméterek értéke valóban felcserélődik a függvényen belül. Csakhogy ezek nem az eredeti változók, hanem azok másolatai.

Nézzük meg a teljes programot:

```
#include <stdio.h>

void csere(int a, int b) {
    int seged = a;
    a = b;
    b = seged;
}

int main() {
    int x = 3;
    int y = 7;

    csere(x, y);

    printf("x = %i, y = %i\n", x, y);

    return 0;
}
```

A program kimenete: `x = 3, y = 7`

Vagyis az eredeti változók nem cserélődtek fel. A csere függvény csak a saját `a` és `b` paramétereit módosította, a `main` függvényben található `x` és `y` változók érintetlenek maradtak.

Ebből fontos következtetés adódik: ha egy függvénynek csak számolnia kell valamit az átadott értékek alapján, akkor az érték szerinti paraméterátadás megfelelő. Ha azonban azt szeretnénk, hogy a függvény az eredeti változókat módosítsa, akkor nem elég azok értékét átadni. A változók címét kell átadnunk.

A C nyelvben a *cím szerinti paraméterátadás* mutatók segítségével valósítható meg. Ilyenkor nem a változók értékét adjuk át a függvénynek, hanem a memóriacímüket. A függvény ezen a címen keresztül eléri és módosítani tudja az eredeti változókat.

A helyes cserefüggvény így néz ki:

```
void csere(int *a, int *b) {
    int seged = *a;
    *a = *b;
    *b = seged;
}
```

A paraméterlistában az `int *a` és `int *b` azt jelenti, hogy a függvény két `int` típusú változó címét várja. A függvényen belül a `*a` és `*b` kifejezések már nem magukat a címeket jelentik, hanem az adott címeken található értékeket.

A teljes program:

```
#include <stdio.h>

void csere(int *a, int *b) {
    int seged = *a;
    *a = *b;
    *b = seged;
}

int main() {
    int x = 3;
    int y = 7;

    csere(&x, &y);

    printf("x = %i, y = %i\n", x, y);

    return 0;
}
```

Most a hívásban nem `x` és `y` szerepel, hanem `&x` és `&y`. Az `&` címképző operátor segítségével az eredeti változók címét adjuk át a függvénynek. A `csere` függvény ezeken a címeken keresztül módosítja az eredeti változókat, ezért a program kimenete már ez lesz: `x = 7, y = 3`

[i] Megjegyzés. A „cím szerinti paraméterátadás” kifejezés a C nyelvben pontosítást igényel. Valójában ilyenkor is érték szerinti átadás történik: a függvény a változó címének másolatát kapja meg. Ez a címérték azonban ugyanarra a memóriaterületre mutat, ahol a hívó függvény változója található, ezért a hívott függvény ezen a címen keresztül módosítani tudja az eredeti változót. *Szemléltetésül*: ha valaki odaadja nekünk a Mona Lisa másolatát, akkor azt szabadon átfesthetjük, bekeretezhetjük vagy összefirkálhatjuk, de ettől az eredeti festmény a Louvreban nem változik meg. Ha viszont nem a festmény másolatát kapjuk meg, hanem a Louvre címének másolatát és a pontos teremszámot, akkor el tudunk jutni az eredeti Mona Lisához. Maga a cím is csak egy másolat a kezünkben, de ugyanarra az eredeti tárgyra mutat. Ha ott módosítanánk a festményt, akkor már az eredeti változna meg.

▶ Összegzés

A függvények segítségével a program részfeladatokra bontható. Egy függvény deklarációja megmondja a fordítónak, hogy milyen függvény létezik; a definíció megadja a működését; a hívás pedig konkrét adatokkal végrehajtatja azt. Az egyszerű paraméterátadás érték szerint történik, vagyis a függvény másolatokkal dolgozik. Ha azt szeretnénk, hogy egy függvény az eredeti változókat módosítsa, akkor azok címét kell átadnunk, és mutatóparamétereken keresztül kell dolgoznunk velük.

11.3. Tömbök átadása függvényeknek

11.3.1. Egydimenziós tömbök mint függvényparaméterek

A 6. fejezetben már találkoztunk egydimenziós tömbökkel. A 8.2. alfejezetben megtudtuk azt is, hogy egy tömb neve sok helyzetben mutatóként kezelhető, és fordítva: egy pointer, amely egy tömb első elemének címét tárolja, használható a tömb neveként, vagyis indexeléssel elérhetők rajta keresztül a tömb elemei. Ez a tudás előfeltétele annak, hogy megértsük, miként történik a tömbök átadása függvényeknek.

A tömbök átadása alapértelmezés szerint címen keresztül történik. Ez azt jelenti, hogy nem a teljes tömb másolódik át a függvénybe, hanem a függvény a tömb első elemének címét kapja meg, pontosabban ennek a címnek a másolatát. Ezt a címet viszont a függvény a tömb neveként használhatja.

Az alábbi példában a `main` függvényben szereplő `t` egy valódi tömbnév. A függvény `t` paramétere viszont valójában egy pointer, amelybe bemásolódik a tömb első elemének címe, és amelyet a függvény tömbnévként tud használni.

```
#include <stdio.h>

int osszeg(int t[], int n);

int main() {
    int t[5] = {3, 8, 1, 6, 4};

    printf("Osszeg: %i\n", osszeg(t, 5));

    return 0;
}

int osszeg(int t[], int n) {
    int s = 0;

    for (int i = 0; i < n; i++) {
        s += t[i];
    }

    return s;
}
```

A függvényben alkalmazott `t[i]` indexelés ugyanazokat az elemeket éri el, amelyek a `main` függvényben deklarált tömbben találhatók.

[i] Megjegyzés. Tömbparaméter esetében a következő két írásmód a függvény fejrészében lényegében ugyanazt jelenti:

```
int osszeg(int t[], int n);
```

és

```
int osszeg(int *t, int n);
```

Mindkét esetben a függvény egy `int` típusú elemekre mutató pointert kap. Az `int t[]` alak azért kényelmesebb kezdő szinten, mert jobban kifejezi, hogy a paramétert a függvényben tömbként fogjuk használni. A mutatós alak viszont pontosabban mutatja, mi történik a háttérben.

[i] Megjegyzés. A `sizeof` operátor jól megmutatja a `main` függvényben szereplő tömbnév és a függvényparaméterként szereplő tömbnév közötti különbséget. A `main` függvényben a `sizeof(t)` az egész tömb méretét adja meg bájtban, vagyis egy 5 elemű `int` tömb esetében 5 `int` méretét. Ezzel szemben az `osszeg` függvényben a `t` paraméter már pointerként viselkedik, ezért ott a `sizeof(t)` egy pointer méretét adja meg, nem az eredeti tömb méretét.

Ezzel összhangban a `main` függvényben használható a `sizeof(t) / sizeof(t[0])` képlet a tömb elemszámának meghatározására, az `osszeg` függvényben viszont nem.

[i] Megjegyzés. A függvény paraméterének nem kötelező ugyanazt a nevet adni, mint a hívó függvényben szereplő tömbnek. Sőt, tanulási szempontból néha hasznos más nevet választani, mert így jobban látszik, hogy a `main`-beli tömbnév és a függvénybeli paraméternév két külön név. Például az `osszeg(t, 5)` hívásban a `t` tömbnév a tömb első elemének címére konvertálódik, és ez a címérték bemásolódhat egy `v` nevű mutatóparaméterbe is. A függvényben ekkor a `v[i]` ugyanazokat az elemeket éri el, mint a `main` függvényben a `t[i]`.

[i] Megjegyzés. Mivel a függvény csak a tömb első elemének címét kapja meg, ebből önmagában nem tudja megállapítani, hogy hány elemből áll a tömb. Ezért a tömb mellé külön átadjuk az elemszámot is. Az `osszeg` függvényben az `n` paraméter mondja meg, hogy hány elemet kell feldolgozni.

11.3.2. Kétdimenziós tömbök átadása függvényeknek

A kétdimenziós tömböket a C nyelv sorok egymás után következő sorozataként tárolja. Egy `int m[10][20];` deklaráció például azt jelenti, hogy az `m` tömb 10 elemből áll, és minden eleme egy 20 egész számot tartalmazó egydimenziós tömb. Másképpen fogalmazva: a kétdimenziós tömb tekinthető egydimenziós tömbök tömbjének.

Ennek megfelelően az `m` tömbnév nem egyetlen `int` elemre mutató pointerként viselkedik, hanem az első sorra mutató pointerként. Mivel egy sor ebben az esetben 20 darab `int` elemből áll, a megfelelő mutatótípus: `int (*p)[20];`. Ez azt jelenti, hogy `p` olyan pointer, amely egy 20 elemű `int` tömbre mutat.

Ezért kétdimenziós tömb függvénynek való átadásakor az oszlopszámnak szerepelnie kell a paraméter típusában. Ha a tömb egy `sor × oszlop` méretű mátrixot tárol (sor ≤ 10, oszlop ≤ 20), akkor a függvény prototípusa az alábbi lehet:

```
int osszeg2D(int m[][20], int sor, int oszlop);
```

A fenti deklarációban az `m` paraméter típusában az első méret elhagyható, a második azonban nem. A fordítónak ugyanis tudnia kell, hány elem alkot egy sort, hogy az `m[i][j]` kifejezés címét helyesen tudja kiszámítani.

Ekvivalens írásmód:

```
int osszeg2D(int (*m)[20], int sor, int oszlop);
```

Kezdő szinten az `int m[][20]` alak szemléletesebb, mert jobban emlékeztet a kétdimenziós tömbre. A mutatós alak viszont pontosabban mutatja, hogy a függvény valójában az első sor címét kapja meg.

Gyakorló feladatok

Az alábbi feladatokban az egydimenziós és kétdimenziós tömbök függvénynek való átadását gyakorolhatjuk. Minden feladatnál megadtuk a megírandó függvény deklarációját. A feladat a függvénydefiníció elkészítése, vagyis a függvény törzsének megírása.

Minden megoldáshoz írj egy `main` függvényt is, amelyben beolvasod vagy inicializálod a szükséges adatokat, meghívod az adott függvényt, majd kiírod az eredményt. A kétdimenziós tömbös példákban feltételezzük, hogy a mátrix legfeljebb 20 oszlopos: `#define MAXO 20`

- Írj függvényt egy egydimenziós tömbben tárolt számsor kiírására.

```
void sor_kiir(int t[], int n);
```

- Írj függvényt egy számsor legnagyobb elemének meghatározására.

```
int sor_max(int t[], int n);
```

- Írj függvényt egy számsor megfordítására helyben.

```
void sor_megfordit(int t[], int n);
```

- Írj függvényt, amely egy növekvően rendezett tömbben keres binárisan egy adott elemet. A függvény térítse vissza az `x` érték első előfordulásának indexét, ha az elem megtalálható a tömbben, és `-1` értéket különben.

```
int keres_rendezett(int t[], int n, int x);
```

Írj függvényt egy mátrix kiírására.

```
void matrix_kiir(int m[][MAXO], int sor, int oszlop);
```

- Írj függvényt annak eldöntésére, hogy egy négyzetes mátrix szimmetrikus-e a főátlóra nézve.

```
int szimmetrikus_e(int m[][MAXO], int n);
```

- Írj függvényt annak eldöntésére, hogy van-e nyeregpont a mátrixban. Egy mátrixelem nyeregpont, ha saját sorában minimális, saját oszlopában pedig maximális.

```
int nyeregpont_van_e(int m[][MAXO], int sor, int oszlop);
```

11.4. BBC szál: Sudoku-ellenőrző



A Sudoku kapcsán a BBC filmje egy mély algoritmikai kérdésre mutat rá: vannak olyan problémák, amelyeknél egy megadott megoldás helyességét könnyű ellenőrizni, miközben mélységes rejtély, hogy a megoldás megkeresésére létezik-e általánosan hatékony algoritmus (11.4. ábra).

Más szóval: ha valaki elénk tesz egy kitöltött Sudoku-táblát, viszonylag gyorsan meg tudjuk vizsgálni, hogy szabályos-e. Ha viszont egy üres vagy részben kitöltött táblát kapunk, akkor a megoldás megtalálása már sokkal nehezebb feladat lehet.



11.4. ábra. *Pillanatkép a BBC dokumentumfilm Sudokuról szóló részéből.*

Mi most az ellenőrzés oldaláról közelítjük meg a problémát: írjunk függvényeket, amelyek ellenőrzik, hogy egy kitöltött Sudoku-tábla helyes-e. A programnak tehát nem megoldania kell a rejtvényt, hanem azt kell eldöntenie, hogy a megadott `9 × 9`-es táblázat megfelel-e a Sudoku szabályainak.

Egy kitöltött Sudoku-tábla akkor helyes, ha:

- minden sorban pontosan egyszer szerepelnek az `1`-től `9`-ig terjedő számok;

- minden oszlopban pontosan egyszer szerepelnek az `1`-től `9`-ig terjedő számok;
- minden `3 × 3`-as résztáblában pontosan egyszer szerepelnek az `1`-től `9`-ig terjedő számok.

Ez a feladat jól illik a függvények témaköréhez, mert a teljes ellenőrzés természetes módon bontható részfeladatokra. Nem egyetlen hosszú programrészben kell gondolkodnunk, hanem kisebb, jól elnevezett ellenőrző függvényekben.

Legyen a Sudoku-tábla egy kétdimenziós, `9 × 9`-es egész tömb: `int tabla[9][9];`

A központi ellenőrző függvény feladata az, hogy eldöntse, az egész tábla helyes-e.

```
bool sudoku_helyes(int t[][9]);
```

Ez a függvény nem maga végzi el az összes részletellenőrzést, hanem leosztja a munkát háromféle kisebb függvénynek:

```
bool sor_helyes(int t[][9], int sor);
bool oszlop_helyes(int t[][9], int oszlop);
bool blokk_helyes(int t[][9], int kezdosor, int kezdooszlop);
```

A `sor_helyes` függvény egy adott sor helyességét ellenőrzi. A sor paraméter azt adja meg, hogy a tábla hányadik sorát kell vizsgálni. Az `oszlop_helyes` függvény hasonló szerepet tölt be, de egy adott oszlopra vonatkozik. A `blokk_helyes` függvény egy `3 × 3`-as résztábla ellenőrzéséért felel. Mivel egy blokkot legegyszerűbben a bal felső sarkával azonosíthatunk, a függvény két paramétert kap: a blokk kezdősorát és kezdőoszlopát. A Sudoku-táblában a `3 × 3`-as blokkok bal felső sarkai a következő pozíciókban találhatók:

```
(0, 0)   (0, 3)   (0, 6)
(3, 0)   (3, 3)   (3, 6)
(6, 0)   (6, 3)   (6, 6)
```

▦ **Algoritmikai mintázat.** Egy sor, egy oszlop vagy egy `3 × 3`-as blokk akkor helyes, ha minden eleme az `1..9` intervallumba esik, és nincs benne ismétlődés. Ennek ellenőrzésére használhatunk egy segédtömböt:

```
int volt[10] = {0};
```

A `volt[k]` értéke azt jelezheti, hogy találkoztunk-e már a `k` számmal. Kezdetben minden érték 0. Amikor egy számot vizsgálunk, először ellenőrizzük, hogy valóban 1 és 9 közé esik-e. Ha nem, a részlet hibás. Ha igen, megnézzük,

szerepelt-e már korábban. Ha a `volt[k]` értéke már 1, akkor ismétlődést találtunk. Ha még 0, akkor beállítjuk 1-re, és folytatjuk az ellenőrzést.

A feladat tehát a fenti függvények definícióinak megírása. Figyelj arra, hogy a `sudoku_helyes` függvény ne maga ellenőrizzen minden elemet közvetlenül, hanem használja fel a `sor_helyes`, `oszlop_helyes` és `blokk_helyes` függvényeket. Így a program szerkezete tükrözni fogja a Sudoku szabályainak természetes felbontását.



AI-csepp

Kérdezd meg az AI-t!

- Milyen táblaméretek számítanak szabályos Sudoku-szerkezetnek?
- Fejtsd ki azt a gondolatot, hogy „vannak olyan problémák, amelyeknél egy megadott megoldás helyességét könnyű ellenőrizni, miközben mélységes rejtély, hogy a megoldás megkeresésére létezik-e általánosan hatékony algoritmus”!

11.5. AlgoRythmics szál: buborékrendezés



A rendezés a programozás egyik alapvető algoritmikai feladata. Adatok rendezésére akkor van szükségünk, amikor egy számsorozatot, névsort, eredménylistát vagy bármilyen más adathalmazt áttekinthetőbbé, kereshetőbbé vagy könnyebben feldolgozhatóvá szeretnénk tenni.

A rendező algoritmusok jelentős része összehasonlításokra épül: két elemet összehasonlítunk, majd az eredmény alapján eldöntjük, hogy maradhatnak-e a helyükön, vagy cserére van szükség. Ezeket nevezzük összehasonlítás alapú rendezéseknek.

A cél minden rendező algoritmusnál ugyanaz, de a stratégia sokféle lehet. Az AlgoRythmics rendezéseket bemutató videói ezt különösen jól szemléltetik: ugyanazt a problémát többféle mozgásmintával, többféle „tánccal” lehet megoldani.

▦ **Algoritmikai mintázat.** Az egyik legismertebb ilyen stratégia a buborékrendezés (11.5. ábra). A módszer szomszédos elempárokat hasonlít össze, és ha rossz sorrendben találja őket, felcseréli. Egy menet során a vizsgált szakasz nagyobb elemei fokozatosan jobbra haladnak, és a szakasz legnagyobb eleme végül „kibuborékozik” a szakasz végére.

Az AlgoRythmics buborékrendezés-videójában jól megfigyelhető, hogy minden menet az első elempárral kezdődik, majd balról jobbra halad. A táncosok mindig szomszédos pozíciókat hasonlítanak össze: ha két szomszéd rossz sorrendben áll, helyet cserélnek; ha jó sorrendben állnak, a menet halad tovább.

Ha csak annyit használunk ki, hogy egy menet végére a vizsgált szakasz legnagyobb eleme a helyére kerül, akkor minden következő menetet eggyel rövidebbre

vehetünk. Az első menet után az utolsó elem már a helyén van, a második menet után az utolsó két elem, és így tovább.



11.5. ábra. *Pillanatképek a buborékrendezést illusztráló csángó néptáncból*

A videóban azonban egy ennél finomabb, optimalizált változat jelenik meg. Minden menetben figyeljük, hol történt az utolsó csere. Az utolsó csere utáni elemek már rendezett helyzetben vannak egymáshoz képest, ezért a következő menetnek elegendő csak eddig a pontig tartania. Ha pedig egy teljes menetben egyetlen csere sem történik, akkor a sorozat már rendezett, és az algoritmus befejezhető.

A feladatunk az, hogy írjuk meg a buborékrendezést megvalósító függvényt. Most azonban nem egyetlen hosszú függvényben gondolkodunk, hanem a rendezést kisebb részfeladatokra bontjuk.

A teljes rendezést a `buborek_rendez` függvény végzi. Ez több menetből áll. Egy menet során a program balról jobbra halad az aktuálisan vizsgált szakaszon, összehasonlítja a szomszédos elemeket, és ha rossz sorrendben állnak, felcseréli őket.

Ezt az egy menetet külön segédfüggvényre bízzuk. Nevezzük ezt `buborekoztat` függvénynek. A `buborekoztat` feladata az, hogy egy adott határig végigmenjen a tömbön, elvégezze a szükséges szomszédos cseréket, majd visszatérítse az utolsó csere helyét. Ez az érték azért fontos, mert a következő menetnek már csak eddig a pontig kell tartania.

A cserét pedig egy harmadik függvény, a már ismert `csere` végzi el. Így a rendezés három egymásra épülő függvény együttműködéséből áll:

```
buborek_rendez
    buborekoztat
        csere
```

A függvények deklarációi:

```
void csere(int *a, int *b);
int buborekoztat(int *t, int hatar);
void buborek_rendez(int *t, int n);
```

A `csere` függvény két egész változó címét kapja meg, és felcseréli a címeken található értékeket.

A `buborekoztat` függvény a `t` tömböt (mint pointert, lásd fentebb) és az aktuális `hatar` értéket kapja meg. A függvény a `0..hatar` szakaszon dolgozik, vagyis az egymás melletti elempárokat `t[0], t[1]`, majd `t[1], t[2]`, és így tovább, egészen `t[hatar - 1], t[hatar]` párig vizsgálja. Ha cserére kerül sor, megjegyzi a csere helyét. A visszatérési értéke az utolsó csere helye.

A `buborek_rendez` függvény vezérli a meneteket. Kezdetben az egész tömböt vizsgáljuk, ezért a határ `n - 1`. Ezután minden menetet a `buborekoztat` függvény végez el, és visszaadja, hogy meddig kell a következő menetben elmenni. Ha egy menetben nem történt csere, az utolsó csere helye 0 marad, így a rendezés befejeződik.

A központi rendezőfüggvény definíciója:

```
void buborek_rendez(int *t, int n) {
    int hatar = n - 1;

    while (hatar > 0) {
        hatar = buborekoztat(t, hatar);
    }
}
```

A segédfüggvények definícióinak megírása az olvasóra marad.



AI-csepp

Kérdezd meg az AI-t!

• Miért fordulhat elő az AlgoRythmics buborékrendezés-videójában, hogy egy-egy menet végén nem fordul meg az összes szakaszvégi elem, amely szemmel láthatóan már a helyén van? Hogyan szemlélteti ez a jelenség a „számítógép vakon rendez” fogalmat?

• Egy algoritmus hatékonyságát gyakran azzal mérjük, hogy legjobb, illetve legrosszabb esetben hány alapműveletet végez. Rendezés esetén ilyen alapművelet lehet például az összehasonlítás és a csere. Az optimalizált buborékrendezés hány összehasonlítást, illetve hány cserét végez egy n elemű, már eleve szigorúan növekvő sorozaton? És hányat egy n elemű, szigorúan csökkenő sorozaton?

11.6. Hiúz szál: hódok és viperák



A 2015-ös Bebras-verseny magyarországi, Hódítsd meg a biteket! változatában szerepelt egy feladat hódokról és viperákról (ELTE 2026). E feladat ihlette a fejezet Hiúz szálon megfogalmazott kihívását.

Egy úszóversenyen hódok és viperák indultak (11.6. ábra). Minden versenyző pontszámot kapott; a nagyobb pontszám jobb eredményt jelent. A pontszámokat egy `n` elemű tömbben tároljuk, mégpedig a résztvevők jelentkezési sorrendjében.



11.6. ábra. *Utolsókból lesznek az elsők*

A versenyről a következőket tudjuk:
- egyetlen vipera sem ért el jobb eredményt, mint bármelyik hód;
- volt legalább egy vipera–hód holtverseny;
- nem volt hód–hód holtverseny.

Írj függvényt, amely meghatározza, hány hód vett részt a versenyen.

```
int hodok_szama(int *pont, int n);
```

Gondold át, hogy a `hodok_szama` függvény mely részfeladatokat oszthatná tovább segédfüggvényeknek.



AI-csepp

Kérdezd meg az AI-t!

- Milyen tipikus hibák képzelhetők el a „Hódok és viperák” feladat szövegének értelmezésekor?

• Hogyan változtatná meg a „Hódok és viperák” feladat megoldási algoritmusát, ha tudjuk, hogy nagyon sok vipera, de kevés hód vett részt a versenyen?

11.7. ECN szál: hegylánc a mátrixban



A 2024-es ECN verseny egyik feladata, a *Climbers*, egy hegygerinc felismerésére épül. A feladat egy négyzetes mátrixot ad meg, és azt kell eldöntenünk, hogy a mátrix sorai együtt szabályos hegygerincet alkotnak-e.

A feladatban először egy *hegysorozat* fogalmát kell megértenünk. Egy számsorozat akkor hegysorozat, ha pontosan egy csúcsa van, a csúcs előtt monoton növekszik, a csúcs után pedig monoton csökken. A csúcs olyan elem, amely szigorúan nagyobb a közvetlen szomszédainál, ha ezek léteznek. Belső elem esetén tehát mindkét szomszédjánál nagyobbnak kell lennie; szélső elem esetén elegendő, ha az egyetlen létező szomszédjánál nagyobb.

A mátrix akkor tartalmaz szabályos hegygerincet, ha két feltétel teljesül:

- a mátrix minden sora hegysorozat;
- az egymást követő sorok csúcsai szomszédosak: vagy ugyanabban az oszlopban vannak, vagy eggyel balra, illetve eggyel jobbra tolódnak.

A feladat abban áll, hogy írjuk meg a `hegygerinc_e` függvényt, amelynek keze alá dolgozik a `csucs_helye` segédfüggvény.

```
bool hegygerinc_e(int m[][MAXN], int n);
int csucs_helye(int sor[], int n);
```

A `csucs_helye` visszatérési értéke kettős szerepet tölt be: jelzi, hogy az adott sor egyáltalán hegysorozat-e, és ha igen, megadja a csúcs helyét. Ha a sor nem hegysorozat, a függvény `-1` értékkel térjen vissza.

A `hegygerinc_e` függvény soronként kérje a `csucs_helye` segédfüggvény segítségét. Ha bármelyik sor nem hegysorozat, vagy ha két egymást követő sor csúcsai nem szomszédosak, akkor a hegygerinc „elromlik”: az ellenőrzés azonnal leállhat, és a függvény `false` értékkel térhet vissza. Ha minden sor rendben van, és a csúcsok megfelelően kapcsolódnak egymáshoz, akkor a függvény `true` értéket térítsen vissza.



AI-csepp

Az eredeti versenyfeladatban `n` értéke akár 2500 is lehet. Egy 2500 × 2500 méretű `int` mátrix eltárolása nagyjából 2500 × 2500 × 4 byte = 25 000 000 byte memóriát igényel, vagyis körülbelül 25 MB-ot. Ez egy átlagos C környezetben

még elfogadható lehet, az eredeti versenykiírás azonban mindössze 128 KB memóriakorlátot adott meg.

Kérdezd meg az AI-t! Hogyan lehet a *Climbers* feladatot memóriatakarékosabban megoldani?

►► Zárógondolat

Ebben a fejezetben átléptünk egy fontos határt. Eddig többnyire kész függvényeket használtunk: a `printf` kiírt, a `scanf` beolvasott, a `sqrt` négyzetgyököt számított. Most azonban nemcsak felhasználói voltunk a függvényeknek, hanem készítői is.

Megtanultuk, hogy egy jól körülhatárolható részfeladatot kiemelhetünk a program fő gondolatmenetéből, nevet adhatunk neki, paramétereken keresztül adatokat adhatunk át, majd visszatérési értékkel vagy módosított változókon keresztül eredményt kaphatunk tőle. A függvény így nem pusztán kényelmi eszköz, hanem a program szerkezetének egyik legfontosabb építőköve.

A függvények használata tehát nemcsak a kód ismétlődését csökkenti, hanem segít gondolkodni is: rákényszerít arra, hogy megfogalmazzuk, pontosan mi egy részfeladat bemenete, mi az eredménye, és hogyan kapcsolódik a nagyobb egészhez. Ettől válik a programozás valóban szervezett problémamegoldássá.

A fejezet végére azt is láttuk, hogy a függvények nem elszigetelt programdarabok. Egy függvény hívhat más függvényeket, azok újabb segédfüggvényekre támaszkodhatnak, és így a program egymásra épülő részfeladatok hálózatává válik. Ezen a ponton már természetes módon merül fel a következő kérdés: mi történik akkor, ha egy függvény nem egy másik függvénynek osztja le a részfeladatot, hanem saját magának?

Erre a különös, első látásra talán meglepő lehetőségre épül a következő fejezet témája: a rekurzió. A rekurzióban egy függvény önmagát hívja meg, de nem körkörös tévedésből, hanem azért, mert a megoldandó probléma kisebb, hasonló szerkezetű részproblémákra bontható. Így a függvényekről tanultak nem lezárulnak, hanem egy új gondolkodási forma felé vezetnek tovább.

12. Rekurzió: amikor a függvény önmagát hívja

Az előző fejezetben megtanultuk, hogy a programot nem kell egyetlen hosszú utasítássorként megírnunk. Egy összetett feladatot részfeladatokra bonthatunk, a részfeladatokat pedig függvényekre bízhatjuk. Láttuk, hogy egy függvény meghívhat egy másik függvényt, sőt az is támaszkodhat további függvényekre.

Most egy első látásra különös kérdést teszünk fel: meghívhatja-e egy függvény saját magát? A válasz: igen. Ezt nevezzük *rekurziónak*.

A rekurzió gondolata matematikából sem ismeretlen. Vannak mennyiségek, amelyeket úgy adunk meg, hogy visszavezetjük őket ugyanannak a mennyiségnek egy kisebb változatára. A faktoriális például így is felírható:

```
5! = 5 · 4!
4! = 4 · 3!
3! = 3 · 2!
2! = 2 · 1!
1! = 1
```

Itt minden lépés ugyanarra a mintára épül: az `n!` kiszámítását visszavezetjük az `(n - 1)!` kiszámítására. A lánc azonban nem folytatódhat örökké: szükség van egy kezdőpontra, egy közvetlenül ismert értékre. Ez itt az `1! = 1`. A rekurzív gondolkodás programozásban is hasonló elven működik.



Ebben a fejezetben a *rekurzív függvények* kerülnek fókuszba. Ezek sok esetben elegáns megoldást adnak olyan problémákra, amelyek természetes módon kisebb, hasonló szerkezetű részfeladatokra bonthatók.

Gondoljunk például egy hagymára. A hagyma héjakból áll: ha lefejtünk róla egy réteget, alatta egy kisebb, de hasonló szerkezetű hagymát találunk. A folyamat addig ismételhető, amíg el nem jutunk a legbelső, már tovább nem bontható részhez. A rekurzió is hasonló gondolatot követ: egy feladatból leválasztunk egy réteget, a maradékot pedig ugyanannak a feladatnak egy kisebb változataként kezeljük.

12.1. Rekurzió: a ciklus alternatívája

Ha egy ismétlődő feladatot kell megoldanunk, eddig többnyire ciklusokban gondolkodtunk. A ciklus lényege az volt, hogy ugyanazt az utasítássort újra és újra végrehajtottuk, amíg a ciklusfeltétel teljesült. Ezt nevezzük *iteratív*

megközelítésnek. Ha ezt a feladatot egy függvényre bízzuk, akkor úgy is mondhatjuk: a függvény ciklus segítségével oldja meg az ismétlést.

A *rekurzív megközelítés* más utat választ. Ilyenkor a függvény nem ciklussal ismétel, hanem – a hagymás hasonlattal élve – „bevállal" csupán egy réteget a feladatból, a maradék részfeladatra pedig, amely hasonló, de már kisebb, meghívja önmagát. Ezt nevezzük *rekurzív hívásnak*.

Szemléletesen elképzelhetjük úgy, mintha a függvény létrehozna önmagából egy új példányt, egy „klónt". A klón persze ugyanazon szabályok szerint működik, mint az őt meghívó példány: bevállal egy réteget, és a maradék feladatot továbbadja egy újabb függvénypéldánynak. Mindez addig folytatódik, amíg a feladat tovább nem bonthatóvá redukálódik. Az a példány, amelyik ezt kapja meg, már nem hoz létre további példányt, hanem közvetlenül megoldja a rá jutó feladatot.

Egy iteratív hagymaevő függvény tehát így gondolkodna:

```
amíg van még hagyma:
    egyél meg egy réteget
```

Egy rekurzív hagymaevő függvény forgatókönyve ezzel szemben:

```
ha a hagyma még többrétegű:
    egyél meg egy réteget
    a maradék hagymára hívd meg önmagad
különben:
    edd meg az egyfalásnyi hagymát
```

Az iteratív megoldásban a függvénynek egyetlen példánya dolgozik, és ez ismétli a műveletet ciklusban. A rekurzív megoldásban több függvénypéldány jön létre: mindegyik elvégez valamennyit a munkából, majd a maradékot továbbadja. Így a példányok mintegy szétosztják maguk között az elvégzendő feladatot.

A rekurzív függvények esetében a hívások láncolata addig folytatódik, amíg a feladat triviálissá nem válik, azaz olyan egyszerűvé, hogy az aktuális példány már közvetlenül meg tudja oldani. Ezt nevezzük *alapesetnek*. Azt az ágat pedig, amelyben a függvény önmagát hívja meg egy kisebb részfeladatra, *rekurzív ágnak* nevezzük.

A kettőnek szorosan össze kell tartoznia. A rekurzív ágnak úgy kell csökkentenie a feladatot, hogy előbb-utóbb biztosan eljussunk az alapesethez. Ha ez nem történik meg, akkor a végtelen ciklus rekurzív párját kapjuk: a függvény újra és újra meghívja önmagát, de soha nem jut el a megálláshoz. Ezt nevezzük végtelen rekurziónak.

[i] Megjegyzés. Vegyük észre, hogy bár az utolsó példány másképpen viselkedik, mint az előzőek, mégis mindegyik – ahogy a klónoktól el is várható – ugyanazt

a forgatókönyvet követi: a kapott feladat méretétől függően választ a rekurzív vagy az alapeseti viselkedésmód között. A függvény forgatókönyvét tehát továbbra is csak egyszer írjuk meg, de rekurzív hívások révén több függvénypéldány hajtja végre.

12.2. A rekurzió oda-vissza útja

Rekurzív híváskor ugyanaz történik, mint bármely más függvényhíváskor: a hívó függvénypéldány várakozik, amíg a hívott függvénypéldány be nem fejezi a munkáját. A különbség csak az, hogy itt a hívó és a hívott ugyanannak a függvénynek két különböző példánya.

Amikor tehát az első példány meghívja a másodikat, az első várakozni kezd. Amikor a második meghívja a harmadikat, a második is várakozik, és így tovább. A hívások láncolata addig mélyül, amíg el nem jutunk az alapesetig. Ott már nem történik újabb rekurzív hívás, ezért a legutolsó példány befejeződik, és visszatér az őt meghívó példányhoz. Ezután a várakozó példányok fordított sorrendben folytatják, majd fejezik be a munkájukat:

```
első példány
    második példány
        harmadik példány
            negyedik példány
                alapeset
            visszatérés a harmadik példányhoz
        visszatérés a második példányhoz
    visszatérés az első példányhoz
```

A rekurzív hívások tehát először egyre mélyebbre vezetnek, amíg el nem érik az alapesetet. Ezután a függvénypéldányok visszafelé, fordított sorrendben fejezik be a munkájukat.

Ezt jól szemlélteti egy lépcsőházas példa. Képzeljük el, hogy egy őr a földszintről indul, és fel kell mennie az 5. emeletre valamit leellenőrizni. Minden szinten ugyanazt a szabályt követi:

- amikor megérkezik egy szintre, felkapcsolja a lámpát;
- ha még nem ért fel az 5. emeletre, továbbmegy egy szinttel feljebb;
- amikor visszatér az adott szintre, lekapcsolja a lámpát.

Vagyis ugyanabban a rekurzív függvényben a munka egyik része a rekurzív hívás előtt, a másik része a rekurzív hívás után történik.

```
ha még nem vagyunk az 5. emeleten:
    kapcsold fel a lámpát ezen a szinten
```

```
    folytasd ugyanezt egy szinttel feljebb
    kapcsold le a lámpát ezen a szinten
különben:
    kapcsold fel a lámpát az 5. emeleten
    végezd el az ellenőrzést
    kapcsold le a lámpát az 5. emeleten
```

Ha a földszintet 0-val jelöljük, a hívások sorrendje így képzelhető el:

```
szint(0)
    szint(1)
        szint(2)
            szint(3)
                szint(4)
                    szint(5)
```

A lámpák felkapcsolása a hívások sorrendjében történik:

```
0. szint: lámpa fel
1. szint: lámpa fel
2. szint: lámpa fel
3. szint: lámpa fel
4. szint: lámpa fel
5. szint: lámpa fel
```

A lámpák lekapcsolása viszont visszatérés közben, fordított sorrendben történik:

```
5. szint: lámpa le
4. szint: lámpa le
3. szint: lámpa le
2. szint: lámpa le
1. szint: lámpa le
0. szint: lámpa le
```

A teljes eseménysor tehát:

```
0. szint: lámpa fel
1. szint: lámpa fel
2. szint: lámpa fel
3. szint: lámpa fel
4. szint: lámpa fel
```

```
5. szint: lámpa fel
5. szint: ellenőrzés
5. szint: lámpa le
4. szint: lámpa le
3. szint: lámpa le
2. szint: lámpa le
1. szint: lámpa le
0. szint: lámpa le
```

Ez a példa jól mutatja, hogy egy rekurzív függvényben kétféle helyen végezhetünk munkát:

```
rekurzív hívás előtt  -> odaúton, a hívások sorrendjében
rekurzív hívás után   -> visszaúton, a visszatérések sorrendjében
```

Például, ha a `printf` utasítás a rekurzív hívás előtt áll, akkor az értékeket a hívások sorrendjében írjuk ki. Ha a `printf` utasítás a rekurzív hívás után áll, akkor előbb eljutunk az alapesetig, és a kiírás visszatérés közben történik.

Ezt szemléltetik az alábbi függvények, amelyek a paraméterként kapott számsorozatot előre, illetve fordított sorrendben írják ki:

```
void kiir_elore(int *t, int n) {
    if (n > 1) {
        printf("%i ", *t);
        kiir_elore(t + 1, n - 1);
    } else {
        printf("%i ", *t);
    }
}

void kiir_vissza(int *t, int n) {
    if (n > 1) {
        kiir_vissza(t + 1, n - 1);
        printf("%i ", *t);
    } else {
        printf("%i ", *t);
    }
}
```

[i] Megjegyzés. A függvény fejlécében most az `int *t` alakot használjuk. Ez azt jelzi, hogy a `t` paraméter egy egész számra mutató pointer. A hívás pillanatában ez mindig az éppen feldolgozandó szakasz első elemére mutat. Ezért `*t` az aktuális szakasz első elemét jelenti. A `t + 1` kifejezés a következő elem címét

adja meg, vagyis a rekurzív hívás már a számsorozat eggyel rövidebb, következő elemtől induló szakaszával dolgozik.

12.3. Rekurzív függvények, amelyek értéket számítanak ki

Az eddigi példákban a rekurzív függvények nem számítottak ki visszatérési értéket, hanem valamilyen műveletet hajtottak végre: például felkapcsoltak és lekapcsoltak egy képzeletbeli lámpát, vagy kiírták egy tömb elemeit előre, illetve visszafelé.

Most olyan rekurzív függvényeket vizsgálunk, amelyek kiszámított eredményt adnak vissza. Ezekben a visszatérési értékeknek különösen fontos szerepük van: a kisebb részfeladat eredménye visszajut az őt meghívó függvénypéldányhoz, amely ezt összekapcsolja a saját munkarészével.

12.3.1. Amikor már van rekurzív képlet

Vannak feladatok, amelyeknél a rekurzív megoldás szinte készen áll előttünk, mert a feladatot eleve rekurzív képlettel is megadhatjuk. Ilyen a faktoriális, amire a bevezető bekezdésekben hivatkoztunk. A matematikából ismert rekurzív képlet a faktoriálisra az alábbi:

```
ha n > 1:
    n! = n · (n - 1)!
különben:
    1! = 1
```

Ez szinte közvetlenül átírható C függvénnyé:

```
int faktorialis(int n) {
    if (n > 1) {
        return n * faktorialis(n - 1);
    } else {
        return 1;
    }
}
```

A `faktorialis(5)` hívás először kisebb részfeladatokra bontja a számítást:

```
faktorialis(5)
    faktorialis(4)
        faktorialis(3)
```

```
faktorialis(2)
    faktorialis(1)
```

A `faktorialis(1)` már az alapeset, ezért 1 értékkel tér vissza. Ezután a korábban várakozó függvénypéldányok fordított sorrendben kapcsolják hozzá a saját munkarészüket:

```
faktorialis(1) = 1
faktorialis(2) = 2 * 1 = 2
faktorialis(3) = 3 * 2 = 6
faktorialis(4) = 4 * 6 = 24
faktorialis(5) = 5 * 24 = 120
```

Itt szépen látszik, hogy a rekurzív hívás eredményét a hívó példány még felhasználja. A `faktorialis(5)` nem maga számolja ki közvetlenül a `4!` értékét, hanem ezt a kisebb részfeladatot a `faktorialis(4)` hívásra bízza. Amikor annak eredménye visszatér, megszorozza 5-tel.

Az is jól látszik ebből a példából, ahogyan a függvénypéldányok szétosztják maguk között az `5!` kiszámításához szükséges négy szorzást. A `faktorialis(5)` az 5-tel, a `faktorialis(4)` a 4-gyel, a `faktorialis(3)` a 3-mal, a `faktorialis(2)` pedig a 2-vel való szorzást „vállalja be”. Ezek a szorzások azonban nem a hívások sorrendjében, hanem visszatérés közben, fordított sorrendben történnek meg. Először eljutunk az alapesetig, majd a példányok sorban visszatérnek, és mindegyik elvégzi a saját szorzását.

[i] Megjegyzés. Ne tévesszen meg bennünket, hogy a `return` utasításban a szorzás jele a rekurzív hívás előtt szerepel. Ez nem ugyanaz az eset, mint amikor a `printf` különálló utasításként állt a rekurzív hívás előtt. A szorzást a függvény csak akkor tudja elvégezni, amikor a `faktorialis(n - 1)` hívás már visszatért, és ismertté vált az eredménye. A kifejezés kiértékelése során tehát előbb meg kell kapnunk a kisebb részfeladat eredményét, és csak ezután lehet azt megszorozni `n`-nel.

12.3.2. Számjegyenkénti feldolgozás

Nem minden rekurzív megoldás indul ismert matematikai képletből. Gyakran elég felismerni, hogy a feladat természetes módon kisebb, hasonló részfeladatra bontható.

Ilyen típusfeladat egy szám számjegyenkénti feldolgozása. Például számítsuk ki egy természetes szám számjegyeinek összegét.

A `4725` szám utolsó számjegye `5`, a maradék szám pedig `472`. Ezért a feladat így bontható:

```
szamjegyosszeg(4725) = 5 + szamjegyosszeg(472)
```

Majd ugyanígy:

```
szamjegyosszeg(472) = 2 + szamjegyosszeg(47)
szamjegyosszeg(47) = 7 + szamjegyosszeg(4)
szamjegyosszeg(4) = 4
```

Az alapeset az, amikor a szám már egyjegyű. Ekkor a számjegyek összege maga a szám.

```
int szamjegyosszeg(int n) {
    if (n >= 10) {
        return n % 10 + szamjegyosszeg(n / 10);
    } else {
        return n;
    }
}
```

Ebben a függvényben

- a saját munkarész az utolsó számjegy leválasztása: `n % 10`
- a kisebb részfeladat pedig a maradék szám számjegyösszegének kiszámítása: `szamjegyosszeg(n / 10)`
- a kettőt a visszatérési értékben kapcsoljuk össze:
  `return n % 10 + szamjegyosszeg(n / 10);`

Ez a gondolat sok hasonló számjegyenkénti feladatra alkalmazható. Például rekurzívan meghatározhatjuk egy szám számjegyeinek számát, megkereshetjük a legnagyobb számjegyét, eldönthetjük, hogy tartalmaz-e adott számjegyet, vagy kiszámíthatjuk a számjegyeinek szorzatát.

12.3.3. Számsorozat elemenkénti feldolgozása

Hasonló gondolkodásmód működik tömbben tárolt számsorozatoknál is. Ha egy sorozat első `n` elemével kell dolgoznunk, akkor külön kezelhetjük az első elemet, a maradék `n - 1` elemet pedig ugyanannak a feladatnak egy kisebb változataként foghatjuk fel.

Például számítsuk ki egy tömb első `n` elemének összegét.

```
int tomb_osszeg(int *t, int n) {
    if (n > 1) {
        return *t + tomb_osszeg(t + 1, n - 1);
    } else {
```

```
            return *t;
        }
    }
```

Itt is a `t` pointer az aktuálisan feldolgozandó szakasz első elemére mutat. Ezért `*t` az aktuális első elem. A `t + 1` pedig a következő elem címét adja meg, vagyis a rekurzív hívás már az eggyel rövidebb, következő elemtől kezdődő szakaszon dolgozik.

Például ha:

```
int t[5] = {8, 3, 6, 2, 9};
```

akkor a gondolatmenet:

```
tomb_osszeg({8, 3, 6, 2, 9}, 5)
= 8 + tomb_osszeg({3, 6, 2, 9}, 4)
= 8 + 3 + tomb_osszeg({6, 2, 9}, 3)
= 8 + 3 + 6 + tomb_osszeg({2, 9}, 2)
= 8 + 3 + 6 + 2 + tomb_osszeg({9}, 1)
= 8 + 3 + 6 + 2 + 9
= 28
```

Ez a minta sok más tömbös feladatnál is használható. Például rekurzívan megkereshetjük a sorozat legnagyobb elemét, megszámolhatjuk a páros elemeket, eldönthetjük, hogy szerepel-e benne egy adott érték, vagy megvizsgálhatjuk, hogy minden eleme megfelel-e egy feltételnek.

A példák célja nem az, hogy minden lehetséges rekurzív helyzetet felsoroljunk. Ez lehetetlen is lenne. A fontos a gondolkodásmód felismerése:

```
Mi az a rész, amelyet most közvetlenül kezelni tudok?
Mi marad utána, ami ugyanilyen típusú, de kisebb feladat?
Hogyan kapcsolódik össze a saját munkarészem a kisebb feladat
eredményével?
Mi az az alapeset, ahol már nincs szükség újabb rekurzív hívásra?
```

Ezekre a kérdésekre nagyon sokféle feladatnál ugyanazzal a rekurzív szemlélettel válaszolhatunk.

[i] Megjegyzés. Ebben az alfejezetben minden rekurzív hívás legfeljebb egy újabb közvetlen függvénypéldányt hozott létre. Alább találkozunk olyan rekurzív megoldásokkal is, amelyekben az aktuális példány több kisebb részfeladatra bontja a problémát, és több rekurzív hívást indít. Erre épül az *oszd meg és uralkodj*

módszer, amely különösen fontos szerepet játszik például hatékony rendezési és keresési algoritmusokban.



AI-csepp – tipikus hibák
Kérd meg az AI-t, hogy szándékosan vétsen tipikus hibákat a fentebb tárgyalt rekurzív feladattípusok témakörében. Ezután javítsd ki a hibás kódokat, majd kérd meg az AI-t, hogy értékelje a javításaidat.

12.4. AlgoRythmics szál: keresés és rendezés rekurzívan



A korábbi fejezetekben már találkoztunk olyan algoritmusokkal, amelyek ismétlődő lépésekből állnak. A bináris keresésnél például mindig megfeleztük a keresési tartományt, a buborékrendezésnél pedig újra és újra végigmentünk a sorozat egy részén. Ezeket eddig ciklusokkal fogalmaztuk meg. Most nézzük meg, hogyan gondolhatjuk újra őket rekurzívan.

12.4.1. Bináris keresés rekurzívan

A bináris keresés különösen jól illik a rekurzív gondolkodáshoz. Ha a keresett elem nem a középső helyen van, akkor a keresés nem a teljes sorozatban folytatódik, hanem annak egyik felében. Vagyis a feladat ugyanaz marad – keresünk egy elemet egy rendezett szakaszban –, csak a szakasz lesz kisebb.

A rekurzív függvény deklarációja:

```
int binaris_keres(int t[], int bal, int jobb, int x);
```

A függvény a `t[bal..jobb]` szakaszban keresi az `x` értéket. Ha megtalálja, visszatér az indexével, különben `-1` értéket ad vissza.

```
int binaris_keres(int t[], int bal, int jobb, int x) {
    if (bal > jobb) {
        return -1;
    }

    int kozep = (bal + jobb) / 2;

    if (t[kozep] == x) {
        return kozep;
    } else if (x < t[kozep]) {
```

```
            return binaris_keres(t, bal, kozep - 1, x);
        } else {
            return binaris_keres(t, kozep + 1, jobb, x);
        }
    }
```

Ebben a példában a rekurzív hívás mindig legfeljebb egy újabb hívást indít, és a keresési tartomány minden lépésben nagyjából megfeleződik.

⏱ Időbonyolultság. A bináris keresés időbonyolultsága abból adódik, hogy az algoritmus minden lépésben megfelezi a vizsgált tartományt. Ha kezdetben (n) elem között keresünk, akkor (k) lépés után legfeljebb ($n/2^k$) elem marad. A keresés akkor ér véget, amikor ez az érték legfeljebb 1, vagyis amikor ($n/2^k \leq 1$). Ebből következik, hogy ($k \geq \log_2 n$), tehát a bináris keresés legrosszabb esetben körülbelül ($\log_2 n$) összehasonlítást végez. Ennek megfelelően az időbonyolultsága ($O(\log n)$).

12.4.2. Buborékrendezés rekurzívan

Az előző fejezetben az AlgoRythmics buborékrendezés-videója alapján megfigyeltük, hogy egy menet során a szomszédos elemeket hasonlítjuk össze, és ha rossz sorrendben vannak, felcseréljük őket. Most az algoritmusnak azt az egyszerű változatát tekintjük, amelyben egyetlen menet az aktuálisan vizsgált szakasz legnagyobb elemét juttatja a szakasz végére. Ha ez megtörtént, akkor a legnagyobb elem már a helyén van, és elegendő a maradék, eggyel rövidebb szakaszt rendezni.

Ebből rekurzív gondolat is készíthető.

```
rendezd a sorozat első n elemét:
    buborékoltasd ki a legnagyobb elemet a szakasz végére
    rendezd rekurzívan az első n - 1 elemet
```

A központi rekurzív függvény:

```
void buborek_rendez_rek(int t[], int n) {
    if (n > 1) {
        buborekoztat(t, n);
        buborek_rendez_rek(t, n - 1);
    }
}
```

A `buborekoztat` függvény feladata egyetlen buborékrendezési menet végrehajtása az első `n` elemen. Ennek megírása az olvasó feladata.

```
void buborekoztat(int t[], int n);
```

A rekurzív változatban minden függvénypéldány egy menetet „vállal be”: a legnagyobb elemet a szakasz végére juttatja, majd a maradék `n - 1` elem rendezését a következő példányra bízza.

Ez a megoldás jól szemlélteti a rekurziót, de nem tekinthető hatékonyabbnak az iteratív buborékrendezésnél. Inkább azt mutatja meg, hogy egy korábban ciklusokkal megfogalmazott algoritmus rekurzívan is újraírható.

12.4.3. Amikor a rekurzió természetes: összefésüléses rendezés

Vannak rendezések, amelyeknél a rekurzív megközelítés nemcsak lehetséges, hanem kifejezetten természetes. Ilyen például az *összefésüléses rendezés*, amely az *oszd-meg-és-uralkodj* gondolatára épül. A feladatot több kisebb részfeladatra bontjuk, ezeket rekurzívan megoldjuk, majd az eredményeket valamilyen módon összekapcsoljuk.

▦ **Algoritmikai mintázat.** Az összefésüléses rendezés alapgondolata:

```
ha a szakasz legfeljebb egy elemű:
    már rendezett
különben:
    oszd két részre
    rendezd rekurzívan a bal oldali részt
    rendezd rekurzívan a jobb oldali részt
    fésüld össze a két rendezett részt
```

A rekurzív függvény váza:

```
void osszefesuleses_rendez(int t[], int bal, int jobb) {
    if (bal < jobb) {
        int kozep = (bal + jobb) / 2;

        osszefesuleses_rendez(t, bal, kozep);
        osszefesuleses_rendez(t, kozep + 1, jobb);

        osszefesul(t, bal, kozep, jobb);
    }
}
```

Az `osszefesul` függvény megírása az olvasó feladata.

```
void osszefesul(int t[], int bal, int kozep, int jobb);
```

Ebben a rendezésekben már nemcsak egyetlen közvetlen rekurzív hívás keletkezik. Egy függvénypéldány két kisebb részfeladatot is létrehoz: a bal és a jobb oldali szakasz rendezését. Ezért ez a példa előkészíti az oszd meg és uralkodj módszert.

Időbonyolultság

A rendezések hatékonysága nagyon különböző lehet. A buborékrendezés egyszerű változata legrosszabb esetben – például amikor a sorozat csökkenő sorrendben van, és növekvő sorrendbe kell rendezni – négyzetes időigényű. Ilyenkor az első menetben `n - 1`, a másodikban `n - 2`, majd egyre kevesebb szomszédos elempárt kell összehasonlítani:

```
(n - 1) + (n - 2) + ... + 2 + 1 = n(n - 1) / 2
```

Ez nagyjából n^2 lépést jelent. Csökkenő sorrendből növekvőbe rendezéskor minden ilyen összehasonlítás cserével is jár, tehát ugyanennyi csere történik.

Az összefésüléses rendezés időigénye ezzel szemben `n log n`. A rekurzív felezések egy bináris felbontási fát hoznak létre, amelynek magassága körülbelül `log n`. Minden szinten összesen lineáris munkát végzünk, mert az összefésülések együtt a teljes sorozat elemeit dolgozzák fel. Így a teljes munka nagyságrendje:

```
n · log n
```

Ezért nagy adatok rendezésére rendszerint nem a buborékrendezést használjuk, hanem hatékonyabb rendezőalgoritmusokat, például összefésüléses rendezést vagy más `n log n` időigényű módszereket, amelyeket későbbi fejezetekben mutatunk be.

[i] Megjegyzés. Az összefésüléses rendezés az *oszd meg és uralkodj* módszer példája. Ilyenkor minden lépésben több kisebb részfeladatra bontjuk a feladatot. Az egyszerűbb rekurziók esetében viszont minden lépésben csak egyetlen kisebb részfeladatra hívódik meg a függvény. Ezt nevezhetjük *csökkentsd és uralkodj* gondolkodásmódnak.

A bináris keresés például minden lépésben megfelezi a keresési tartományt. Itt a feladat mérete egy szorzótényezővel csökken: nagyjából `n`-ről `n / 2`-re. A rekurzív buborékrendezés egyszerű változatában viszont minden menetben csak eggyel rövidül a még rendezetlen szakasz: `n` elem helyett már csak `n - 1` elemet kell rendezni.

Mindhárom esetben kisebb feladatok felé haladunk, de más ütemben és más szerkezetben.



AI-csepp

Kérdezd meg az AI-t!

• Mennyi a lineáris keresés, a bináris keresés, a buborékrendezés és az összefésüléses rendezés időbonyolultsága legjobb, legrosszabb és átlagos esetben? Kérj segítséget a képletek értelmezéséhez is, ha szükséges.

• Hogyan lehet iteratívan implementálni az összefésüléses rendezést? Miben különbözik ez a rekurzív változattól, és mikor lehet indokolt az iteratív megoldást használni? Nézd meg az idekapcsolódó AlgoRythmics videót is!

12.5. BBC szál: PageRank – amikor a fontosság mások fontosságától függ



A weben az oldalak hivatkozásokkal kapcsolódnak egymáshoz. Egy oldalra mutathat sok másik oldal, és maga az oldal is mutathat további oldalakra. Így a web nem egyszerű lista, hanem hatalmas hálózat. De hogyan dönthetjük el egy ilyen hálózatban, hogy melyik oldal fontosabb a másiknál?

A BBC film a PageRank gondolatát egy focicsapat példájával szemlélteti. Képzeljük el, hogy egy mérkőzésen a játékosok passzolgatnak egymásnak. Ha egy játékos sok labdát kap a társaitól, akkor első közelítésben azt mondhatjuk: fontos szereplője a játéknak. De nem mindegy, kitől kapja a labdát. Ha egy játékos olyan társaktól kap passzokat, akik maguk is sokat vesznek részt a játékban, akkor ezek a passzok nagyobb súlyt képviselnek. Egy játékos „fontossága” tehát nemcsak attól függ, hány passzt kap, hanem attól is, hogy mennyire fontos játékosoktól kapja őket.

Ez már önmagába visszatérő gondolat: egy játékos fontos, ha fontos játékosoktól kap passzokat. Más szóval, a fontosság meghatározása egy hálózaton belül egymásra épül.

Ugyanez a gondolat jelenik meg a weboldalak rangsorolásában is, ugyanis a PageRank eredetileg weboldalak fontosságát próbálta mérni. Egy weboldal annál fontosabbnak tekinthető, minél fontosabb oldalak hivatkoznak rá. A passzok helyére tehát hivatkozásokat, a játékosok helyére pedig weboldalakat képzelhetünk.

A PageRank gondolatát képlettel is felírhatjuk:

```
PR(A) = (1 - d) / N + d · összeg(PR(T) / L(T))
```

Itt `d` egy úgynevezett csillapítási tényező, `N` az oldalak száma, `T` pedig azokat az oldalakat jelöli, amelyek `A` oldalra hivatkoznak.

A képletből jól látszik az önhivatkozó jelleg: egy oldal PageRank-értékét más oldalak PageRank-értékeiből számítjuk ki. Csakhogy ezeknek a más oldalaknak a PageRank-értéke is további oldalak PageRank-értékétől függ.

Ez a gondolat hasonlít a rekurzióhoz: egy értéket hasonló jellegű értékek segítségével határozunk meg. Van azonban egy fontos különbség. A faktoriálisnál a rekurzív hívások egyre kisebb feladatokhoz vezetnek, végül elérjük az alapesetet:

```
5! -> 4! -> 3! -> 2! -> 1!
```

A PageRank hálózatában viszont könnyen körbeérhetünk. Az egyik oldal hivatkozhat a másikra, a másik egy harmadikra, a harmadik pedig vissza az elsőre. Ilyenkor nincs egyszerű „utolsó hagymahéj”, vagyis nincs olyan alapeset, amelynél a számítás magától megállna.

Ezért a PageRanket a gyakorlatban nem egyszerű alapesetes rekurzív függvényként számítjuk ki, hanem iteratív közelítéssel. Kezdetben minden oldal kap valamilyen induló rangot. Ezután újra és újra frissítjük az oldalak rangját annak alapján, hogy milyen rangú oldalak hivatkoznak rájuk. Sok ismétlés után az értékek egyre kevésbé változnak, és kialakul egy stabil rangsor.

A PageRank tehát érdekes módon köti össze a rekurzív és az iteratív gondolkodást. Maga az ötlet önhivatkozó: egy oldal fontossága más oldalak fontosságától függ. A kiszámítás viszont tipikusan iteratív: az értékeket lépésről lépésre közelítjük.

[i] Megjegyzés. A PageRank példája arra is figyelmeztet, hogy az önhivatkozó gondolat nem mindig vezet közvetlenül rekurzív programhoz. Sőt, még olyan feladatoknál is, amelyeknek van alapesetes rekurzív megoldása, célszerű lehet inkább iteratív feldolgozást választani.

A rekurzió sokszor elegáns, rövid és könnyen érthető megoldást ad, különösen akkor, ha a probléma természetes módon kisebb, hasonló szerkezetű részfeladatokra bomlik. Más esetekben viszont a ciklus egyszerűbb, gyorsabb vagy memóriatakarékosabb lehet. Minden rekurzív hívás új függvénypéldányt hoz létre, amelynek adatait a futtatókörnyezetnek nyilván kell tartania. Ha a hívások láncolata nagyon hosszú, ez felesleges többletmunkát vagy akár hibát is okozhat.

Ezért a rekurziót nem csodaszerként, hanem erős gondolkodási és programozási eszközként érdemes kezelni. Vannak helyzetek, amikor kifejezetten ajánlott, mert világosan tükrözi a feladat szerkezetét. Máskor viszont ugyanazt a feladatot jobb ciklussal megoldani. A jó programozó nemcsak azt tudja, hogyan kell rekurziót írni, hanem azt is mérlegeli, mikor érdemes használni.



AI-csepp

Remélhetőleg ez az alfejezet felkeltette a kíváncsiságodat a PageRank algoritmus iránt. Ez tipikus helyzet, amikor az AI-t magad mellé hívhatod

tutorként a további kutakodáshoz. Feltétlenül kérdezz rá az algoritmus történetére is: tanulságos.

12.6. Hiúz szál: kijut-e a hiúz a labirintusból?



Egy hiúz egy labirintusban bolyong. A labirintust négyzetrácsként adjuk meg, ahol minden mező vagy szabad, vagy fal. Ismerjük a hiúz aktuális pozícióját. A labirintusból úgy lehet kijutni, ha a hiúz elér egy szabad mezőt a külső kereten.

A labirintust kétdimenziós egész tömbként tároljuk. A bemenetben a mezők jelentése:

```
0    szabad mező
1    fal
```

Példabemenet:

```
6 7
1 1
1 1 1 1 1 1 1
1 0 0 0 1 0 1
1 0 1 0 0 0 1
1 0 1 0 1 0 0
1 0 0 0 0 0 1
1 1 1 1 1 1 1
```

Az első sor a labirintus méretét adja meg: 6 sor és 7 oszlop. A második sor a hiúz kezdőpozíciója: `1 1`. Ezután következik a labirintus mátrixa. Ebben a példában a kijárat a `3 6` koordinátájú mező.

A rekurzív gondolat természetes: a hiúz a klónjai életre hívása révén deríti ki, hogy van-e kiút számára a labirintusból. Ha az aktuális példány nem éppen egy keretbeli szabad cellán áll, vagyis még nem jutottunk ki, akkor a szomszédos szabad cellákra létrehoz egy-egy további klónt önmagából. Ezeknek a hiúzpéldányoknak az a feladatuk, hogy megmondják, van-e az ő helyükről kivezető út. Ha valamelyikük igennel válaszol, akkor az aktuális helyről is van kiút. Ha egyikük sem talál kijáratot, akkor erről a mezőről nincs kivezető út.

Fontos azonban valamire odafigyelnünk: ahogy egy valódi labirintusban is fennáll a körbejárás veszélye, ugyanez a veszély a rekurzív hívások láncolatában is megjelenik. Ennek elkerülésére elegáns megoldás, ha a rekurzív hívásokat képviselő hiúzpéldányok foglaltnak jelölik azt a cellát, amelyen létrejöttek. Erre használhatjuk a `-1` értéket. Amikor egy példány befejezte a munkáját, mielőtt

felszámolódna, a szóban forgó mezőt újra szabaddá teszi, vagyis a megfelelő cellát visszaállítja `0`-ra.

A függvény deklarációja lehet például:

```
int kijut_e(int lab[][MAXO], int sor, int oszlop, int i, int j);
```

A `lab` a labirintus, `sor` és `oszlop` a mérete, `i` és `j` pedig az aktuális mező koordinátái.

A függvényben minden irányhoz külön változót rendelünk. Ezek azt tárolják, hogy az adott irányból ki lehet-e jutni.

```
int kijut_e(int lab[][MAXO], int sor, int oszlop, int i, int j) {

    if (i == 0 || i == sor - 1 || j == 0 || j == oszlop - 1) {
        return 1;
    }

    lab[i][j] = -1;

    int van_ut_fel, van_ut_le, van_ut_balra, van_ut_jobbra;
    van_ut_fel = van_ut_le = van_ut_balra = van_ut_jobbra = 0;


    if (lab[i - 1][j] == 0) {
        van_ut_fel = kijut_e(lab, sor, oszlop, i - 1, j);
    }

    if (lab[i + 1][j] == 0) {
        van_ut_le = kijut_e(lab, sor, oszlop, i + 1, j);
    }

    if (lab[i][j - 1] == 0) {
        van_ut_balra = kijut_e(lab, sor, oszlop, i, j - 1);
    }

    if (lab[i][j + 1] == 0) {
        van_ut_jobbra = kijut_e(lab, sor, oszlop, i, j + 1);
    }

    lab[i][j] = 0;

    return van_ut_fel || van_ut_le || van_ut_balra || van_ut_
jobbra;
}
```

[i] Megjegyzés. Ezt a módszert *backtrackingnek*, magyarul gyakran *visszalépéses keresésnek* nevezzük. Az algoritmus kipróbál egy lehetőséget, ideiglenesen megjelöli a hozzá tartozó állapotot, majd ha az adott irány vizsgálata befejeződött, visszaállítja az állapotot, és más lehetőségeket is megvizsgálhat. Ha a backtracking gondolata felkeltette a kíváncsiságodat, szólítsd magad mellé az AI-t tutorként, és kutassatok tovább együtt!

[i] Megjegyzés. Ez a megoldás úgynevezett *mélységi keresést* végez. A hiúzpéldányok először elindulnak egy irányba, ameddig tudnak, majd ha ott nem találnak kijáratot, a hívások visszatérnek, és másik irányok vizsgálata következik. A mélységi keresésről is érdemes később többet kérdezned az AI-tól.



AI-csepp
Kérdezd meg az AI-t!
• Van-e más megoldás is a végtelen rekurzió elkerülésére azon kívül, hogy foglaltnak jelöljük azokat a cellákat, amelyeken éppen létező hiúzpéldány dolgozik?
• Hogyan kellene módosítani a fenti kódot, ha nemcsak azt szeretnénk eldönteni, hogy létezik-e kijutási út, hanem az összes hurokmentes kiutat ki szeretnénk írni?
• Hogyan változna a megoldás, ha nem akármilyen kijutási utat keresnénk, hanem a legrövidebb kiút érdekelne bennünket?
• Létezik-e hatékonyabb megoldás erre a feladatra, mint az, hogy a rekurzív keresés minden lehetséges irányt végigpróbál?

12.7. ECN szál: függőségek – mit kell előbb létrehozni?



Az ECN verseny 2023-as kiadásának egyik feladata az *AWS CloudFormation* világából merített ötletet (*K – Dependencies*). Egy felhőalapú rendszerben különböző erőforrásokat hozunk létre: szervereket, hálózati elemeket, tárhelyeket és más szolgáltatásokat. Ezek között függőségek lehetnek. Egy erőforrást csak akkor lehet létrehozni, ha azok az erőforrások, amelyektől függ, már léteznek.

Mi most a feladat egyszerűsített változatát vizsgáljuk. Az erőforrásokat nem nevekkel, hanem sorszámokkal jelöljük (`0, 1, 2, 3, ...`), és adott erőforrás létrehozhatóságának ellenőrzésére szorítkozunk.

A függőségeket egy kétdimenziós tömbben tárolhatjuk:

```
int fugg[MAXN][MAXN];
```

A jelentés:

```
fugg[i][j] = 1    az i-edik erőforrás függ a j-ediktől
fugg[i][j] = 0    nincs ilyen függőség
```

Ha például a `3`-as erőforrás létrehozásához előbb szükség van az `1`-es és a `2`-es erőforrásra, akkor azt mondjuk, hogy a `3`-as erőforrás függ az `1`-estől és a `2`-estől, ami abban tükröződik, hogy `fugg[3][1] = 1` és `fugg[3][2] = 1`.

A rekurzív gondolat itt is természetes: egy erőforrás létrehozható, ha minden olyan erőforrás létrehozható, amelytől ő maga függ. Vagyis ha meg akarjuk tudni, létrehozható-e a `3`-as erőforrás, akkor meg kell kérdeznünk ugyanezt az `1`-esről és a `2`-esről is. Ha azoknak is vannak függőségeik, akkor a kérdés továbbadódik még kisebb részfeladatokra.

A függvény deklarációja lehet például:

```
int letrehozhato_e(int fugg[][MAXN], int n, int eroforras, int
allapot[]);
```

Az `allapot` tömb azt jelzi, hogy az egyes erőforrások vizsgálata hol tart:

```
0     még nem vizsgáltuk
-1    éppen vizsgáljuk
1     létrehozható
2     nem hozható létre
```

A `-1` érték különösen fontos. Ha egy erőforrás vizsgálata közben újra eljutunk ugyanahhoz az erőforráshoz, akkor körkörös függőséget találtunk. Például: `1 függ 2-től, 2 függ 3-tól, 3 függ 1-től`. Ilyenkor egyik sem hozható létre elsőként, mert mindegyik vár valamelyik másikra. Ez olyan, mintha három hiúz egymást várná a labirintusban: senki sem tud elindulni (lásd az előző alfejezetet).

A rekurzív függvény egy lehetséges definíciója:

```
int letrehozhato_e(int fugg[][MAXN], int n, int eroforras, int
allapot[]) {

    // az erőforrást egy korábbi ágon létrehozhatónak találtuk
    if (allapot[eroforras] == 1) {
        return 1;
    }

    // az erőforrást egy korábbi ágon már létrehozhatatlannak
találtuk
```

```
    if (allapot[eroforras] == 2) {
        return 0;
    }

    // az erőforrás vizsgálat alatt áll; körkörös függőséget
találtunk
    if (allapot[eroforras] == -1) {
        allapot[eroforras] = 2;
        return 0;
    }

    // az aktuális erőforrást vizsgálat alá helyezzük
    allapot[eroforras] = -1;

    // ellenőrizzük azoknak az erőforrásoknak a
létrehozhatóságát,
    // amelyektől az aktuális erőforrás függ
    for (int i = 0; i < n; i++) {
        if (fugg[eroforras][i] == 1) {
            if (!letrehozhato_e(fugg, n, i, allapot)) {
                allapot[eroforras] = 2;
                return 0;
            }
        }
    }

    // ha minden szükséges erőforrás létrehozhatónak bizonyult,
    // akkor az aktuális is létrehozható
    allapot[eroforras] = 1;

    return 1;
}
```

Ebben a függvényben az aktuális példány egyetlen erőforrás létrehozhatóságát vizsgálja. Ha az erőforrás más erőforrásoktól függ, akkor ezekhez újabb rekurzív hívások tartoznak. A függvény tehát mintha azt kérdezné: én csak akkor jöhetek létre, ha mindenki létrehozható, akire szükségem van? Ha valamelyik függőségi lánc visszavezet egy már éppen vizsgált erőforráshoz, akkor körkörös függőséget találtunk, és a válasz nem.

[i] Megjegyzés. Az eredeti ECN-feladatban az erőforrásoknak szöveges neveik voltak, és egy megadott létrehozási sorrendről kellett eldönteni, hogy helyes-e. Mi most a feladat rekurzív magját emeltük ki: azt vizsgáljuk, hogyan terjed végig a „létrehozható-e?” kérdés a függőségi láncokon.

[i] Megjegyzés. Ez a feladat rokonságban áll a gráfokkal. Az erőforrásokat csúcsoknak, a függőségeket pedig irányított éleknek tekinthetjük. A körkörös függőség ilyenkor irányított körnek felel meg.

[i] Megjegyzés. Ha a függőségi hálózat nem tartalmaz körkörös függőséget, akkor az erőforrások mindig sorba rendezhetők úgy, hogy minden erőforrás után következzenek azok, amelyek tőle függnek. Ezt *topologikus sorrendnek* nevezzük. Egy ilyen sorrend ismeretében az erőforrások egymás után, minden további vizsgálat nélkül létrehozhatók.



AI-csepp

Kérdezd meg az AI-t!

• Hogyan lehetne az egyszerűsített feladat megoldását közelebb vinni az eredeti ECN-feladat követelményéhez, ahol egy megadott erőforrás-sorrendet kell ellenőrizni, hogy megvalósítható-e?

►► Zárógondolat

Ebben a fejezetben egy első látásra különös gondolattal ismerkedtünk meg: egy függvény meghívhatja önmagát. A rekurzió azonban nem varázslat, hanem gondolkodásmód. Akkor működik jól, ha egy feladat természetes módon kisebb, hasonló szerkezetű részfeladatokra bontható.

Láttuk, hogy minden rekurzív megoldásnak szüksége van alapesetre: olyan helyzetre, amikor a függvény már nem hívja meg önmagát, hanem közvetlenül választ ad. Ugyanilyen fontos a rekurzív ág is, amely a feladatot kisebbé teszi, és közelebb visz az alapesethez. Ha ez nem történik meg, a rekurzió végtelen hívásláncca válik.

A rekurzió ereje abban rejlik, hogy nem mindig nekünk kell egyszerre átlátni az egész feladatot. Elég pontosan megfogalmaznunk, mit végezzen el az aktuális függvénypéldány, milyen kisebb részfeladatokat adjon tovább, és mikor álljon meg. A hívások láncolata, majd a visszatérések sorrendje elvégzi a többit.

Fontos azonban azt is megjegyezni, hogy a rekurzió nem minden esetben a legjobb megoldás. Sok feladat iteratívan is megoldható, néha egyszerűbben, gyorsabban vagy memóriatakarékosabban. A jó programozó nem azért választ rekurziót, mert lehet, hanem azért, mert a feladat szerkezete szinte felkínálja.

A rekurzióval egy újabb fontos lépést tettünk az algoritmikus gondolkodás felé. Most már nemcsak utasításokat, ciklusokat és függvényeket látunk, hanem feladatok egymásba ágyazódó szerkezetét is: kisebb problémákat, amelyek együtt oldanak meg egy nagyobbat.

13. Karakterek és karakterláncok

Eddigi programjainkban többnyire számokkal dolgoztunk. Egész számokat és valós számokat olvastunk be, műveleteket végeztünk rajtuk, sorozatokat, tömböket és mátrixokat dolgoztunk fel. A valóságban azonban a programok nagyon gyakran nemcsak számadatokat kezelnek, hanem szövegeket is: neveket, azonosítókat, jelszavakat, üzeneteket, parancsokat, fájlneveket vagy akár teljes mondatokat.

A C nyelvben a szövegek kezelése szorosan kapcsolódik ahhoz, amit a korábbi fejezetekben már megtanultunk a típusokról, a tömbökről, mutatókról és a függvényekről. A 9. fejezetben már találkoztunk a `char` típussal. Akkor azt mondtuk, hogy a `char` különleges szerepet tölt be: egyrészt kis egész értékek tárolására használható, másrészt karakterek kezelésére. Azt is jeleztük, hogy tömbökkel kombinálva, vagyis `char[]` alakban, a C nyelvben a szövegek alapvető reprezentációját adja.



Ebben a fejezetben a visszatérünk karakterek és karakterláncok témához, és részletesebben megvizsgáljuk, hogyan kezeli a C nyelv ezeket. A fejezet fő gondolata egyszerű, de nagyon fontos: C-ben a szöveg karakterekből álló tömb, amelyet egy speciális lezáró karakter zár le. Ezt a speciális lezáró karaktert így írjuk: `'\0'`.

A `'\0'` nem azonos a `'0'` karakterrel. A `'0'` a nulla számjegy karaktere, míg a `'\0'` a nulla értékű karakter, amely a karakterlánc végét jelzi.

13.1. Karakterek: amikor a betű mögött szám áll

A `char` típusról korábban már volt szó, ezért itt csak röviden idézzük fel a legfontosabb tudnivalókat. Erre azért van szükség, mert a karakterláncok megértéséhez először azt kell látnunk, hogyan kezeli a C nyelv az egyes karaktereket.

A `char` típus neve arra utal, hogy gyakran karakterek tárolására használjuk. A memóriában azonban nem maga a betű „képe” tárolódik, hanem egy számkód. Az angol ábécé betűi, a számjegyek és sok írásjel például ASCII-kóddal ábrázolható.

Szemléltetésül: a `char c = 'A';` változódefiníció után, ha `c` értékét `%c` formátumjellel íratjuk ki, akkor `A` jelenik meg, ha viszont `%i` formátumjellel, akkor `65`, hiszen az `'A'` karakter ASCII-kódja `65`.

Mivel a karakterek mögött számkódok állnak, összehasonlíthatjuk őket. Példaként tekintsük az alábbi rövid függvényeket. Ezek rendre eldöntik, hogy egy karakter kisbetű-e, nagybetű-e vagy számjegy-e; meghatározzák egy számjegy karakter számértékét; illetve nagybetűvé alakítanak egy betűkaraktert.

```
int kisbetu_e(char c) {
    return c >= 'a' && c <= 'z';
}

int nagybetu_e(char c) {
    return c >= 'A' && c <= 'Z';
}

int szamjegy_e(char c) {
    return c >= '0' && c <= '9';
}

int szamjegy_ertek(char c) {
    return c - '0';
}

char nagybetusit(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - ('a' - 'A');
    } else {
        return c;
    }
}
```

Egyetlen karakter beolvasásához a `%c` formátumjelet használhatjuk:

```
char c;
scanf("%c", &c);
```

A `%c` azonban minden karaktert beolvas, a szóközt és az új sor karaktert is. Ez néha meglepetést okozhat, főleg, ha előtte számot olvastunk be. Gyakran hasznosabb így írni:

```
scanf(" %c", &c);
```

A `%c` előtti szóköz azt jelenti, hogy a `scanf` átugorja az üres karaktereket: szóközöket, tabulátorokat és sorvégeket, és csak ezután olvassa be a következő tényleges karaktert.

[i] Megjegyzés. Az itt bemutatott összehasonlítások az ASCII-kódtábla egyszerű szerkezetére épülnek. Az angol ábécé kisbetűi, nagybetűi és a számjegyek egymást követő kódtartományokat alkotnak, ezért működnek az olyan feltételek, mint például `c >= 'a' && c <= 'z'`. A magyar ékezetes betűk azonban nem részei

az eredeti ASCII-kódtáblának, ezért ezekre az itt bemutatott egyszerű vizsgálatok nem alkalmazhatók közvetlenül. Ebben a fejezetben az egyszerűség kedvéért főként az angol ábécé betűivel, a számjegyekkel és az alapvető írásjelekkel dolgozunk.

13.2. Karakterláncok: szöveg mint karaktertömb

A C nyelvben nincs külön „szöveg” alaptípus. A szöveget karakterekből álló tömbként tároljuk.

Például:

```
char s[] = "alma";
```

Ez valójában egy karaktertömb:

```
s[0] = 'a'
s[1] = 'l'
s[2] = 'm'
s[3] = 'a'
s[4] = '\0'
```

A karakterlánc végén tehát egy speciális lezáró karakter áll: `'\0'`. Ez jelzi, hogy a szöveg véget ért. Ezért az `"alma"` karakterlánc tárolásához nem 4, hanem 5 karakterhelyre van szükség. Az ötödik helyen a lezáró `'\0'` található.

13.2.1. Karakterláncok beolvasása és kiírása

Karakterláncot a `%s` formátumjellel olvashatunk be és írhatunk ki.

```
#include <stdio.h>

int main() {
    char nev[50];

    scanf("%s", nev);
    printf("Szia, %s!\n", nev);

    return 0;
}
```

Figyeljük meg, hogy a karakterláncokat nem karakterenként olvassuk be, mint ahogyan korábban a számsorozatok elemeit olvastuk be. Egyetlen `%s` formátumjel segítségével a `scanf` több karaktert is beolvas egymás után.

Azt is vegyük észre, hogy a `scanf` hívásban a tömb neve előtt nem szerepel a címoperátor (`&`). Ennek oka az, hogy a tömb neve mutatóként, azaz az első elem címeként viselkedik. Erről már a tömbök és a függvények kapcsán is beszéltünk.

Fontos továbbá, hogy vigyázzunk a tömb méretére. Ha a `nev` tömb 50 elemű, akkor legfeljebb 49 karakter hosszú szöveget olvashatunk bele, mert a lezáró `'\0'` karakternek is hely kell. Ezt a lezáró karaktert a `scanf` a beolvasott karaktersor végére automatikusan elhelyezi.

Ezért biztonságosabb így írni:

```
scanf("%49s", nev);
```

A `49` azt jelzi, hogy a `scanf` legfeljebb 49 tényleges karaktert olvashat be. Az 50. hely így megmarad a lezáró `'\0'` karakter számára.

A számsorozatok beolvasásánál általában előre megadtuk, hány számot szeretnénk beolvasni. Ha például `n` szám következett, akkor egy ciklus pontosan `n` alkalommal olvasott be egy-egy értéket. Karakterláncok esetén gyakran más a helyzet. Itt nem feltétlenül tudjuk előre, hány karakterből áll majd a beolvasandó szöveg. Ehelyett a beolvasást valamilyen határoló jel állítja meg.

A `%s` formátumjel például az első üres karakterig olvas. Üres karakternek számít többek között a szóköz, a tabulátor és a sorvége. Ezért a `scanf("%49s", nev)` hívás egyetlen szót olvas be, de nem olvas be szóközt tartalmazó teljes nevet vagy mondatot.

Ha a sorvégéig szeretnénk olvasni a `scanf` segítségével, akkor egy trükkös megoldást alkalmazhatunk: a `scanf` úgynevezett *scanset* formáját.

```
scanf("%49[^\n]", sor);
```

Ez legfeljebb 49 karaktert olvas be, vagy addig olvas, amíg el nem éri a sorvége karaktert. A lezáró `'\0'` karakternek itt is helyet kell hagynunk.

[i] Megjegyzés. A `scanf` *scanset* formája azt jelenti, hogy a `%` jel után szögletes zárójelben megadjuk, milyen karaktereket olvashat be a bemenetről. A beolvasás az első olyan karakternél megáll, amely nem tartozik ehhez a halmazhoz. Például a `scanf("%[abc]", s)` csak az `a`, `b` és `c` karakterek közül olvas be. Az első más karakter esetén a beolvasás leáll.

Ha a nyitó szögletes zárójel után közvetlenül a `^` karakter szerepel, akkor a jelentés megfordul: a `scanf` minden olyan karaktert beolvas, amely *nem* tartozik a megadott halmazhoz. Ezért a `scanf("%[^\n]", s)` alak azt üzeni a `scanf`

függvénynek: olvass karaktereket, de állj meg a sorvége karakter előtt. Így valósul meg a sorvégig tartó beolvasás.

[i] Megjegyzés. A `scanf("%49[^\n]", sor);` nem olvassa be magát a sorvége karaktert, csak megáll előtte. Emiatt a sorvége ott marad a bemeneten. Ha ezt a sorvége karaktert el szeretnénk dobni, külön beolvashatjuk tárolás nélkül a `scanf("%*c");` függvényhívással. A `*` azt jelenti, hogy a beolvasott karaktert nem rendeljük hozzá változóhoz. Így a sorvége eltűnik a bemenetről, de nem kerül eltárolásra. Együtt tehát:

```
scanf("%49[^\n]", sor);
scanf("%*c");
```

[i] Megjegyzés. A `%[^\n]` használata akkor működik célszerűen, ha valóban van mit beolvasni a sorvége előtt. Üres sor esetén a `%[^\n]` nem olvas be semmit, és a cél tömb tartalma változatlan marad. Ha azt szeretnénk, hogy üres sor esetén a sor tömb üres karakterláncot tartalmazzon, beolvasás előtt beállíthatjuk: `sor[0]='\0';`

13.2.2. Műveletek karakterláncokkal

A karakterlánc végét a `'\0'` karakter jelzi. Ezért a karakterlánc bejárásakor addig haladunk, amíg el nem érjük ezt a lezáró karaktert.

```
int i = 0;

while (s[i] != '\0') {
    printf("%c\n", s[i]);
    i++;
}
```

Ebből könnyen megírhatjuk a karakterlánc hosszát meghatározó saját függvényünket. A változatosság kedvéért itt for ciklust használunk:

```
int hossz(char s[]) {
    int i;

    for (i = 0; s[i] != '\0'; i++) {
        ;
    }

    return i;
}
```

A ciklus törzse üres, hiszen minden szükséges munka a ciklusfejben történik.

[i] Megjegyzés. Ez az egyik legfontosabb különbség a karaktertömb és a karakterlánc között. A karaktertömb mérete azt mondja meg, hány karakter fér el benne. A karakterlánc hossza viszont azt mondja meg, hány karakter van ténylegesen a lezáró `'\0'` előtt.

A karakterláncokat tehát ugyanúgy bejárhatjuk, mint más tömböket. A különbség az, hogy itt nem feltétlenül a tömb méretét adjuk meg külön paraméterként, hanem a `'\0'` jelzi a végét.

Mivel a karakterlánc valójában karaktertömb, a függvények módosíthatják az eredeti szöveget. Szemléltetésül nézzünk egy függvényt, amely minden kisbetűt nagybetűvé alakít:

```
void nagybetusit_mind(char *s) {
    int i = 0;

    while (s[i] != '\0') {
        if (s[i] >= 'a' && s[i] <= 'z') {
            s[i] = s[i] - 'a' + 'A';
        }
        i++;
    }
}
```

A `char *s` jelölés azt hangsúlyozza, hogy a függvény valójában a karakterlánc első karakterének címét kapja meg. Ugyanezt a paramétert írhatnánk `char s[]` alakban is; függvényparaméterként a két forma ekvivalens. Mivel a függvény a tömb elemeit ezen a címen keresztül éri el, a módosítás az eredeti karakterláncban történik.

Ha a függvényt a

```
char s[] = "Alma a fa alatt";
```

karakterláncra hívjuk meg, akkor

```
nagybetusit_mind(s);
```

után az `s` tartalma ez lesz:

```
ALMA A FA ALATT
```

Külön figyelmet igényel a karakterláncok összehasonlítása és másolása. Mivel a tömbnevek sok helyzetben címként viselkednek, az `s1 == s2` kifejezés azt

vizsgálja, hogy a két karakterlánc ugyanott kezdődik-e a memóriában, nem pedig azt, hogy az `s1` és `s2` címeken kezdődő karakterláncok tartalma azonos-e.

Hasonló módon karakterláncot nem másolhatunk egyszerű értékadással: `s1 = s2`; Tömbök esetén ez nem jelenti a karakterek átmásolását egyik tömbből a másikba.

Ha saját kezűleg szeretnénk megvalósítani ezeket a műveleteket, akkor ciklust kell írnunk a karakterláncok karakterenkénti összehasonlítására, illetve másolására. A jó hír az, hogy léteznek erre kész könyvtári függvények is, amelyeket a következő alfejezet tárgyal.

13.2.3. A <string.h> könyvtár

A C szabványos könyvtára számos alapvető karakterlánc-művelethez kínál függvényeket. Ezek deklarációi a `<string.h>` fejlécállományban találhatók.

A négy legalapvetőbb karakterlánc-kezelő függvény:

```
strlen     karakterlánc hosszának meghatározása
strcmp     két karakterlánc összehasonlítása
strcpy     karakterlánc másolása
strcat     karakterlánc hozzáfűzése egy másikhoz
```

A karakterlánc hosszának meghatározására használhatjuk a `strlen` függvényt:

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

int main() {
    char s[50];

    scanf("%49s", s);

    printf("%i\n", strlen(s));

    return 0;
}
```

A `strlen(s)` a lezáró `'\0'` előtti karakterek számát adja vissza. A lezáró karaktert tehát nem számolja bele a hosszba.

Két karakterlánc tartalmi összehasonlítására a `strcmp` függvény használható:

```
if (strcmp(a, b) == 0) {
    printf("Egyformak.\n");
}
```

A `strcmp(a, b)` visszatérési értéke:

```
0      ha a két karakterlánc azonos
< 0    ha a előbb következik a szótári sorrendben, mint b
> 0    ha a később következik a szótári sorrendben, mint b
```

Karakterlánc másolására a `strcpy` függvény szolgál:

```
strcpy(cel, forras);
```

Ez a `forras` karakterlánc karaktereit átmásolja a `cel` tömbbe, a lezáró `'\0'` karakterrel együtt.

Karakterláncok összefűzésére a `strcat` használható:

```
strcat(cel, forras);
```

Ez a `forras` karakterláncot hozzáfűzi a `cel` karakterlánc végéhez. A művelet után a `cel` tartalmazza az összefűzött karakterláncot.

[i] Megjegyzés. A `strcpy` és a `strcat` használatakor nekünk kell gondoskodnunk arról, hogy a cél tömb elég nagy legyen. Másoláskor hely kell a forrás összes karakterének és a lezáró `'\0'` karakternek is. Összefűzéskor pedig a cél tömbnek az eredeti tartalom, a hozzáfűzött tartalom és a lezáró `'\0'` karakter számára is elegendő helyet kell biztosítania.

A `<string.h>` további hasznos függvényeket is tartalmaz. Például egy karakter első előfordulását a `strchr` függvénnyel kereshetjük meg:

```
char *p = strchr(s, 'a');
```

Ha az `'a'` karakter szerepel az `s` karakterláncban, akkor `p` az első előfordulás címét kapja meg. Ha nem szerepel benne, akkor `p` értéke `NULL` lesz.

Egy rövidebb karakterlánc előfordulását egy hosszabb karakterláncban a `strstr` függvény keresi:

```
char *p = strstr(s, "alma");
```

Ha az `"alma"` előfordul az `s` karakterláncban, akkor `p` az előfordulás kezdetére mutat. Ha nem fordul elő, akkor `NULL` értéket kapunk.

Léteznek olyan függvények is, amelyeknél megadhatjuk, legfeljebb hány karaktert vegyenek figyelembe. Ilyen például a `strncmp`, amely legfeljebb az első `n` karakter alapján hasonlít össze két karakterláncot:

```
if (strncmp(a, b, n) == 0) {
    printf("Az elso n karakter azonos.\n");
}
```

Másolásra és hozzáfűzésre is léteznek korlátozott változatok:

```
strncpy(cel, forras, n);
strncat(cel, forras, n);
```

Ezek legfeljebb `n` karaktert másolnak, illetve fűznek hozzá. Fontos azonban, hogy ezek használata sem mentesít minket a tömbméretek figyelése alól.

[i] Megjegyzés. A `<string.h>` könyvtár ennél jóval több karakterlánc- és memóriakezelő függvényt tartalmaz. Ebben az alfejezetben csak a leggyakrabban használt, kezdő szinten is jól érthető függvényekkel foglalkozunk. Később, összetettebb szövegfeldolgozási feladatoknál érdemes lesz további függvényeket is megismerni.

13.3. Szöveges állomány feldolgozása karakterenként, szavanként és soronként

Karakterláncokat nem csak billentyűzetről olvashatunk be. Gyakran előfordul, hogy a feldolgozandó szöveg állományban található: például egy üzenetben, egy naplófájlban, egy forráskódban vagy egy hosszabb dokumentumban.

Korábban számsorozatoknál sokszor előre tudtuk, hány adatot kell beolvasnunk. Szöveges állományoknál ez általában nincs így. Ilyenkor természetes megállási feltétel az állomány vége. A `scanf` visszatérési értéke megmutatja, sikerült-e újabb adatot beolvasni, ezért állományvégig tartó feldolgozásnál gyakori minta, hogy maga a `scanf` hívás kerül a while feltételébe.

Ha állományból szeretnénk olvasni, használhatjuk a korábban már látott `freopen` függvényt:

```
freopen("szoveg.txt", "r", stdin);
```

Ettől kezdve a `scanf` nem a billentyűzetről, hanem a szoveg.txt állományból olvas.

Karakterenkénti feldolgozás

Ha az állomány minden karakterét fel szeretnénk dolgozni, akkor a `%c` formátumjelet használjuk:

```
char c;

while (scanf("%c", &c) == 1) {
    /* a c karakter feldolgozása */
}
```

Ebben az esetben nem teszünk szóközt a `%c` elé, mert a szóköz azt jelentené, hogy a `scanf` átugorja az üres karaktereket: szóközöket, tabulátorokat és sorvégeket. Szöveges állomány feldolgozásakor ezek is a szöveg részei lehetnek.

Szavankénti feldolgozás

Szavankénti feldolgozáshoz a `%s` formátumjel használható:

```
char szo[50];

while (scanf("%49s", szo) == 1) {
    /* az szo karakterlánc feldolgozása */
}
```

A `%s` az első üres karakterig olvas. Ezért ez a forma akkor hasznos, ha a szöveget szavak sorozataként szeretnénk feldolgozni.

Soronkénti feldolgozás

Ha soronként szeretnénk olvasni, akkor a `scanf` scanset formáját használhatjuk:

```
char sor[100];

while (scanf(" %99[^\n]", sor) == 1) {
    /* a sor feldolgozása */
}
```

A `%99[^\n]` legfeljebb 99 karaktert olvas be, és megáll a sorvége előtt. A formátumsztring elején álló szóköz arra szolgál, hogy a `scanf` átugorja az előző beolvasás után a bemeneten maradt sorvége karaktert.

[i] Megjegyzés. Ez az egyszerű soronkénti változat az üres sorokat átugorja, mert a kezdő szóköz minden üres karaktert elnyel. Ebben az alfejezetben most elegendő azt látnunk, hogy a `scanf` segítségével karakterenként, szavanként és sorvégig is olvashatunk.

Összefoglalva:

```
%c          egyetlen karaktert olvas be
%s          karakterláncot olvas be az első üres karakterig
%[^\n]      karakterláncot olvas be a sorvége karakterig
```

Gyakorló feladatok

- Írj függvényt, amely egy karakterláncban minden adott karaktert egy másik karakterre cserél.

```
void csere_mind(char s[], char mit, char mire);
```

- Írj függvényt, amely eltávolítja egy karakterláncból az összes szóközt.

```
void szokozok_torlese(char s[]);
```

- Írj függvényt, amely eldönti, hogy egy karakterlánc palindrom-e. Egy karakterlánc palindrom, ha balról jobbra és jobbról balra olvasva ugyanaz. Írd meg a függvényt iteratívan és rekurzívan is.

```
int palindrom_e(char s[]);
```

- Írj függvényt, amely eldönti, hogy egy jelszó elég erős-e. Akkor erős a jelszó, ha tartalmaz legalább egy kisbetűt, legalább egy nagybetűt és legalább egy számjegyet.

```
int eros_jelszo_e(char s[]);
```

- Írj függvényt, amely eldönti, hogy egy karakterlánc szabályos C-változónév lehet-e. Egyszerűsített szabály: az első karakter betű vagy _, a további karakterek betűk, számjegyek vagy _ jelek lehetnek.

```
int valtozonev_e(char s[]);
```

- Írj függvényt, amely megszámolja, hány magánhangzó található egy karakterláncban. Az egyszerűség kedvéért csak az angol magánhangzókat vedd

figyelembe: `a, e, i, o, u`, valamint ezek nagybetűs változatait. Írd meg a függvényt iteratívan és rekurzívan is.

```
int maganhangzok_szama(char s[]);
```

- Írj függvényt, amely helyben megfordít egy karakterláncot.

```
void megfordit(char s[]);
```

- Olvass be karakterenként egy szöveget állományvégig, és írd ki minden karakter ASCII-kódját.
- Olvass be szavanként egy szöveget állományvégig, és számold meg, hány szó található benne.
- Olvass be szavanként egy szöveget állományvégig, és írd ki a leghosszabb szót. Feltételezheted, hogy minden szó legfeljebb 49 karakter hosszú.
- Olvass be soronként egy szöveget állományvégig, és minden sorban alakítsd nagybetűssé a kisbetűket.

[i] Megjegyzés. Az állományfeldolgozó feladatoknál használd a fejezetben látott beolvasási mintákat: karakterenként `scanf("%c", &c)`, szavanként `scanf("%49s", szo)`, soronként pedig `scanf(" %99[^\n]", sor)`.



AI-csepp – tipikus hibák
Kérdezd meg az AI-t!
- Melyek a tipikus hibák a karakteres, illetve karakterláncos feladatok megoldásakor?

13.4. AlgoRythmics szál: névsor építése beszúró rendezéssel



Az AlgoRythmics videóiban a beszúró rendezés gondolata különösen szemléletes: az elemeket egyenként illesztjük be a már rendezett rész megfelelő helyére.

Ebben a fejezetben ezt a gondolatot karakterláncokra alkalmazzuk. Képzeljük el, hogy egy szöveges állomány egyszavas neveket tartalmaz. A feladatunk az, hogy az állományból ábécérendbe rendezett névsort készítsünk.

A beszúró rendezést most nem úgy használjuk, hogy előbb minden nevet beolvasunk, majd utólag rendezzük a tömböt. Ehelyett már beolvasás közben építjük a rendezett névsort: amikor egy új név megérkezik, megkeressük a helyét az addig beolvasott, már rendezett nevek között, és oda szúrjuk be.

A bemeneti állomány például ilyen lehet:

```
Kata
Bence
Anna
Dora
Csaba
```

A névsor lépésenként így alakul:

```
Kata

Bence
Kata

Anna
Bence
Kata

Anna
Bence
Dora
Kata

Anna
Bence
Csaba
Dora
Kata
```

A programban a neveket kétdimenziós karaktertömbben tároljuk:

```
#define MAXN 100
#define MAXH 50

char nevek[MAXN][MAXH];
```

A `nevek` tömb legfeljebb `MAXN` nevet tárolhat, és minden név legfeljebb `MAXH - 1` karakter hosszú lehet. Azért csak `MAXH - 1`, mert a lezáró `'\0'` karakter számára is helyet kell hagynunk.

A beolvasás és a beszúrás együtt történik:

```
int rendezve_beolvas(char nevek[][MAXH]) {
```

```
    int n = 0;
    char uj[MAXH];

    while (n < MAXN && scanf("%s", uj) == 1) {
        beszur(nevek, n, uj);
        n++;
    }

    return n;
}
```

A `rendezve_beolvas` függvény minden újonnan beolvasott nevet azonnal beszúr a megfelelő helyre. Az `n` változó azt mutatja, hogy hány név van már a rendezett névsorban.

A beszúrás a beszúró rendezés gondolatát követi. Jobbról balra haladva eltoljuk azokat a neveket, amelyek az új név után következnének az ábécérendben, majd a felszabadult helyre bemásoljuk az új nevet.

```
void beszur(char nevek[][MAXH], int n, char uj[]) {
    int i = n - 1;

    while (i >= 0 && strcmp(nevek[i], uj) > 0) {
        strcpy(nevek[i + 1], nevek[i]);
        i--;
    }

    strcpy(nevek[i + 1], uj);
}
```



13.1. ábra. *Jelenet a beszúró rendezést román néptánccal illusztráló AlgoRythmics videóból*

[i] Megjegyzés. Az itt alkalmazott módszer különbözik a videóban látható változattól. Az aktuális elemet nem „bebuborékoltatjuk” a helyére, hanem eltolásokkal készítünk helyet az új névnek, majd bemásoljuk a felszabadult helyre (13.1. ábra).

Ez a megoldás különösen természetes karakterláncok esetén. Ha minden lépésben szomszédos neveket cserélgetnénk, sok karakterlánc-másolásra lenne szükség. Az eltolásos változatban az új nevet ideiglenesen az `uj` tömbben tartjuk, a szükséges neveket egy-egy hellyel jobbra toljuk, végül az `uj` értékét bemásoljuk a megfelelő helyre. [**i**]

Íme egy lehetséges `main` függvény:

```
int main() {
    char nevek[MAXN][MAXH];
    int n;

    freopen("nevek.txt", "r", stdin);

    n = rendezve_beolvas(nevek);
    kiir(nevek, n);

    return 0;
}
```

A `kiir` függvény megírását az olvasóra hagyjuk.

Ez a változat jól mutatja, miért kapta a beszúró rendezés a nevét: minden új elem a már rendezett rész megfelelő helyére kerül. A fejezet szempontjából az is fontos, hogy az algoritmus ugyanaz marad, csak az adattípus változik. Számok helyett most karakterláncokat rendezünk. Az összehasonlításhoz a `strcmp`, a mozgatáshoz pedig a `strcpy` függvényre van szükség.



AI-csepp

Kérdezd meg az AI-t!

- Mi változna, ha az állomány többszavas neveket tartalmazna, soronként egyet?
- Mennyi a beszúró rendezés időbonyolultsága legjobb és legrosszabb esetben?
- Mennyit nyerünk azzal, hogy az aktuális elemet nem „bebuborékoltatjuk” a helyére, hanem eltolásokkal készítünk helyet neki, majd beszúrjuk?
- Mivel minden lépésben egy már rendezett sorban kell megkeresnünk az új elem helyét, gyorsítható-e a beszúró rendezés bináris keresés alkalmazásával?

13.5. BBC szál: stabil házasság probléma – nevek, listák, párosítások



A BBC-filmben szereplő stabil házasság probléma első látásra nem karakterláncos feladatnak tűnik. Valójában azonban nagyon jól kapcsolódik ehhez a fejezethez, mert a szereplőket nevekkel azonosítjuk, a bemenet pedig preferenciasorrendeket tartalmazhat: ki kit részesít előnyben.

Képzeljünk el két azonos méretű csoportot. A klasszikus megfogalmazásban az egyik csoport tagjai férfiak, a másik csoport tagjai nők. Mindenki sorrendbe állítja a másik csoport tagjait aszerint, hogy kit választana legszívesebben.

Például:

```
Adam: Eva Anna Dora
Bence: Anna Dora Eva
Csaba: Anna Eva Dora

Eva: Bence Adam Csaba
Anna: Adam Csaba Bence
Dora: Csaba Bence Adam
```

A cél olyan párosítás létrehozása, amely stabil. Egy párosítás akkor nem stabil, ha van két ember, akik nem egymással vannak párban, de mindketten jobban választanák egymást a jelenlegi párjuknál. Stabil párosítás esetén ilyen zavaró pár nem létezik.

Ebben a fejezetben a feladat karakterláncos oldalára figyelünk. A bemenetben nevek szerepelnek, a program belsejében viszont kényelmesebb indexekkel dolgozni. Például:

```
Adam   -> 0
Bence  -> 1
Csaba  -> 2
```

és

```
Eva    -> 0
Anna   -> 1
Dora   -> 2
```

A nevek tárolására használhatunk kétdimenziós karaktertömböket:

```
#define MAXN 20
```

```
#define MAXH 30

char ferfiak[MAXN][MAXH];
char nok[MAXN][MAXH];
```

A preferenciasorrendeket már tárolhatjuk egész számokkal. Ha például `ferfi_pref[i][j]` értéke `k`, az azt jelenti, hogy az `i` indexű férfi preferenciasorrendjében a `j`-edik helyen a `k` indexű nő áll.

```
int ferfi_pref[MAXN][MAXN];
int no_pref[MAXN][MAXN];
```

Ehhez szükségünk van egy függvényre, amely egy névhez megkeresi a megfelelő indexet:

```
int nev_indexe(char nevek[][MAXH], int n, char nev[]) {
    for (int i = 0; i < n; i++) {
        if (strcmp(nevek[i], nev) == 0) {
            return i;
        }
    }

    return -1;
}
```



(a)



(b)

13.2. ábra. *(a) Pillanatkép a BBC film stabil házasság problémát illusztráló jelenetéből. (b) Shapley és Roth 2012-ben megkapták a közgazdasági Nobel-emlékdíjat „a stabil allokációk elméletéért és a piactervezés gyakorlati alkalmazásáért”. Gale 2008-ban elhunyt, ezért nem részesülhetett a díjban.*

A stabil házasság algoritmus egyik legismertebb megoldása a Gale–Shapley-algoritmus (13.2. ábra). Ennek gondolata röviden a következő:

```
amíg van szabad férfi, aki még nem kérte meg minden nő kezét:
    a férfi megkéri a számára legkedvezőbb olyan nőt,
    akit még nem kért meg

    ha a nő szabad:
        ideiglenesen elfogadja a kérést
    különben:
        összehasonlítja az új kérőt a jelenlegi párjával,
        és azt tartja meg, akit jobban kedvel
```

A párosítás menet közben változhat. Egy nő először elfogadhat egy ajánlatot, később azonban lecserélheti a párját, ha olyan kérő érkezik, akit a saját listája alapján jobban kedvel.

A döntéshez gyorsan meg kell tudnunk mondani, hogy egy nő két férfi közül melyiket részesíti előnyben. Erre készíthetünk egy rangsoroló táblázatot. Ha `no_rang[no][ferfi]` értéke kisebb, akkor az adott nő jobban kedveli azt a férfit.

```
int jobban_kedveli_e(int no_rang[][MAXN], int no, int uj_ferfi,
int regi_ferfi) {
    return no_rang[no][uj_ferfi] < no_rang[no][regi_ferfi];
}
```

Figyeljük meg, hogy a programban a neveket és a sorszámokat együtt használjuk. A bemenet és a kimenet az ember számára olvasható nevekkel dolgozik, az algoritmus belsejében viszont az indexek teszik egyszerűvé a tömbök kezelését.

A fejezet szempontjából ez a legfontosabb tanulság: a karakterláncok gyakran azonosítóként szerepelnek. Személyeket, városokat, tantárgyakat, erőforrásokat vagy más objektumokat neveznek meg. A program sokszor először ezeket a neveket sorszámokká alakítja, majd a számokkal végzi el az algoritmikus feldolgozást.

[i] Megjegyzés. A stabil házasság probléma teljes megoldása már összetettebb algoritmus. Itt nem az volt a célunk, hogy minden részletét leprogramozzuk, hanem az, hogy meglássuk: a karakterláncok nemcsak szövegként, hanem azonosítóként is fontos szerepet játszanak a programokban.



AI-csepp

A PageRank kapcsán már láttuk, hogy egy rövid filmes példa mögött egész algoritmikus világ nyílhat meg. A stabil házasság problémája is ilyen: elsőre játékos párosítási feladatnak tűnik, valójában azonban fontos gondolatokat rejt döntésekről, sorrendekről, stabilitásról és igazságosságról.

Kérdezd meg az AI-t!

- Hogyan működik a Gale–Shapley-algoritmus a fentebb említett három-három szereplős példán?
- Miért garantált, hogy a Gale–Shapley-algoritmus véget ér?
- Milyen értelemben kedvezhet az algoritmus annak az oldalnak, amelyik a javaslatokat teszi?
- Milyen története van a stabil házasság problémának és a Gale–Shapley-algoritmusnak?
- Milyen valós alkalmazásai vannak ennek az algoritmusnak a házassági példán túl?
- Hogyan lehetne adatszerkezetekkel leírni a stabil házasság problémát `n` férfi és `n` nő esetére?
- Segítsen az AI ezek alapján C nyelven leprogramozni a Gale–Shapley-algoritmust karakterláncként megadott nevekkel és preferenciasorrendekkel.

13.6. Hiúz szál: Caesar-titkosítás



A hiúzok titkos üzeneteket szeretnének küldeni egymásnak. Ehhez egy régi, híres titkosítási módszert használnak: a Caesar-titkosítást.

A módszer lényege egyszerű: az üzenet minden betűjét néhány helylyel eltoljuk az ábécében. Ha az eltolás értéke 3, akkor az A betűből D, a B betűből E, a C betűből F lesz. Az ábécé végén körbeérünk: az X betűből A, az Y betűből B, a Z betűből pedig C lesz. A kisbetűket és a nagybetűket külön kezeljük. A szóközök, számjegyek és írásjelek változatlanul maradnak.

Írj függvényt, amely egy karakterláncot Caesar-módszerrel titkosít!

```
void caesar_titkosit(char s[], int eltolas);
```

Például 3-as eltolással:

```
Bemenet:
Szia, Hiuz!

Kimenet:
Vcld, Klxc!
```

A függvény járja be a karakterláncot a lezáró `'\0'` karakterig. Ha az aktuális karakter kisbetű, tolja el az angol ábécén belül. Ha nagybetű, tolja el a nagybetűk között. Minden más karakter maradjon változatlan.

[i] Megjegyzés. A Caesar-titkosítás történeti szempontból érdekes, és programozási feladatként is hasznos, de mai szemmel nem tekinthető biztonságos módszernek, hiszen könnyen feltörhető.



AI-csepp

Kérdezd meg az AI-t!

- Milyen módszerekkel törhető fel egy Caesar-titkosított szöveg?
- Tervezz és implementálj saját feltörő függvényeket, majd kérd meg az AI-t, hogy tesztelje és értékelje ki őket különböző titkosított szövegeken.
- Milyen más titkosítási módszerek léteznek?

13.7. ECN szál: karakterlánc egyszerűsítése



A 2024-es ECN verseny egyik feladata a karakterláncok egyszerűsítéséről szól (*String Simplification*). A feladatot a versenyen 21 csapatból 17 megoldotta, vagyis a mezőny nagy része megtalálta a helyes megoldást.

Egy karakterlánc bonyolultságát most az határozza meg, hogy hány különböző karakter szerepel benne. Például a `better` karakterláncban négy különböző betű fordul elő: `b`, `e`, `t` és `r`.

Dominik nem szereti a túl bonyolult karakterláncokat, ezért néhány karaktert törölni szeretne úgy, hogy a megmaradó karakterláncban legfeljebb kétféle különböző betű szerepeljen. A kérdés az, hogy legalább hány karaktert kell törölnie.

A bemenetben csak az angol ábécé kisbetűi szerepelnek, ezért elegendő egy 26 elemű gyakoriságtömböt használnunk.

A kulcsgondolat a következő: ha a végén legfeljebb kétféle betű maradhat, akkor érdemes azt a két betűt megtartani, amelyek a legtöbbször fordulnak elő. Minden más betűt törölni kell. Vagyis a törlendő karakterek számát úgy kapjuk meg, hogy a karakterlánc hosszából kivonjuk a két legnagyobb betűgyakoriság összegét.

Például az `assesses` karakterláncban a betűk gyakorisága:

```
a: 1
s: 4
e: 2
```

A két leggyakoribb betű az `s` és az `e`, ezekből összesen 6 darab van. A karakterlánc hossza 7, ezért csak 1 karaktert kell törölni: az `a` betűt.

[i] Megjegyzés. A bemenet mérete kicsi: legfeljebb 100 karakterláncot kapunk, és mindegyik hossza legfeljebb 100 karakter. Emiatt a versenyen sok csapat

rendezéses megoldást alkalmazott. Például rendezhetjük a betűk gyakoriságait, majd kiválaszthatjuk a két legnagyobbat. A 26 elemű gyakoriságtömb miatt ez is bőven elég gyors. Az alábbi megoldás viszont rendezés nélkül, egyetlen bejárással keresi meg a két legnagyobb gyakoriságot.

A megoldást függvényként is megfogalmazhatjuk:

```
int torlendo_karakterek_szama(char s[]);
```

A függvény először megszámolja, hogy melyik betű hányszor szerepel a karakterláncban, majd megkeresi a két legnagyobb gyakoriságot.

```
#include <stdio.h>

#define ABC 26

int torlendo_karakterek_szama(char s[]);

int main() {
    int t;
    char s[101];

    scanf("%i", &t);

    for (int i = 0; i < t; i++) {
        scanf("%100s", s);
        printf("%i\n", torlendo_karakterek_szama(s));
    }

    return 0;
}

int torlendo_karakterek_szama(char s[]) {
    int db[ABC] = {0};
    int hossz = 0;
    int max1 = 0;
    int max2 = 0;

    for (int i = 0; s[i] != '\0'; i++) {
        db[s[i] - 'a']++;
        hossz++;
    }

    for (int i = 0; i < ABC; i++) {
        if (db[i] > max1) {
```

```
                max2 = max1;
                max1 = db[i];
            } else {
                if (db[i] > max2) {
                    max2 = db[i];
                }
            }
        }

        return hossz - max1 - max2;
    }
```

A `db[s[i] - 'a']++` utasítás azt használja ki, hogy az angol kisbetűk egymás után következnek a kódtáblában. Így az `a` betű indexe 0, a `b` betű indexe 1, és így tovább.

Figyeljük meg, hogy nem kell kipróbálnunk minden lehetséges törlést. A feladat lényege nem az, hogy ténylegesen töröljük a karaktereket, hanem az, hogy eldöntsük, mennyit kellene törölni. Ha tudjuk, melyik két betűből van a legtöbb, akkor a válasz már közvetlenül kiszámítható.

[i] Megjegyzés. Ez a feladat jól mutatja, hogy a karakterlánc-feldolgozásban gyakran nem magát a szöveget alakítjuk át, hanem információt gyűjtünk róla: hosszát, karaktereit, gyakoriságait vagy mintázatait vizsgáljuk.



AI-csepp

Kérdezd meg az AI-t!

- Mi lehetett az oka annak, hogy négy csapat nem tudta helyesen megoldani ezt a feladatot?
- Mi lehetett az oka annak, hogy egyes csapatoknak csak többszöri próbálkozás után sikerült helyes megoldást beküldeni?
- Milyen tipikus félreértések fordulhatnak elő ennél a feladatnál?
- Milyen tesztesetekkel lehetne kiszűrni a hibás megoldásokat?

►► Zárógondolat

Ebben a fejezetben a számok világából átléptünk a szövegek világába. Láttuk, hogy C-ben a karakterlánc nem külön varázslatos adattípus, hanem `char` elemekből álló tömb, amelyet a lezáró `'\0'` karakter tesz karakterlánccá.

Megtanultuk, hogyan olvashatunk be és írhatunk ki karaktereket, szavakat és sorokat, hogyan járhatunk be egy karakterláncot a lezáró karakterig, és hogyan módosíthatjuk annak tartalmát helyben. Közben fontos különbségekkel is találkoztunk: nem ugyanaz a `'0'`, a `0` és a `'\0'`; karakterláncokat nem az == operátorral hasonlítunk össze; karakterláncokat nem egyszerű értékadással másolunk.

A fejezet példái azt is megmutatták, hogy a szöveg a programokban sokféle szerepet kaphat. Lehet üzenet, amelyet Caesar-módszerrel titkosítunk; lehet névsor, amelyet ábécérendbe rendezünk; lehet személynév, amelyet egy algoritmus belső indexszé alakít; és lehet versenyfeladat bemenete, amelyből gyakoriságokat számolunk.

A karakterláncok kezelése aprólékos figyelmet igényel. Mindig gondolnunk kell a tömb méretére, a lezáró `'\0'` karakterre, a beolvasás megállási feltételeire és arra, hogy a szóközök, sorvégek, számjegyek és írásjelek is a szöveg részei lehetnek. Ha ezeket tudatosan kezeljük, akkor a karakterláncok nem zavaró kivételek lesznek, hanem erős eszközök: segítségükkel a programjaink emberi nyelven megadott adatokat is képesek feldolgozni.

14. Struktúrák: amikor több adat egy objektumhoz tartozik

Eddig változóink többnyire egyetlen adatot tároltak: egy egész számot, egy valós számot, egy karaktert vagy egy karakterláncot. Tömböket is használtunk már, de a tömb minden eleme azonos típusú volt: például egész számok sorozata, karakterek sorozata vagy karakterláncok tömbje.

A valóságban azonban gyakran több, különböző típusú adat tartozik ugyanahhoz a dologhoz: egy pontnak lehet két koordinátája, egy tanulóhoz tartozhat név és pontszám, egy könyvhöz tartozhat cím, szerző és oldalszám stb. Külön változókban is tárolhatnánk ezeket az adatokat, de így könnyen szétesne az, ami logikailag összetartozik. A C nyelvben a `struct` típus éppen arra való, hogy több adatot egyetlen összetett egységbe fogjunk össze.



Ebben a fejezetben megtanuljuk, hogyan foghatunk össze több, logikailag összetartozó adatot egyetlen egységbe a `struct` segítségével. Látni fogjuk, hogyan hozhatunk létre saját összetett típusokat, hogyan érhetjük el ezek mezőit, és hogyan tárolhatunk sok hasonló objektumot struktúrákból álló tömbben. A fejezet végére már nemcsak különálló számokban, karakterláncokban és tömbökben gondolkodunk, hanem olyan adatcsomagokban is, mint egy pont, egy tanuló, egy könyv vagy egy versenyző.

14.1. Az első struktúra

Kezdjük egy egyszerű példával. Egy síkbeli pontot két egész számmal írhatunk le: az `x` és az `y` koordinátával.

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

Ez még nem változódefiníció, hanem egy új típus leírása. Azt mondjuk meg vele, hogy egy `struct Pont` típusú adat két mezőből áll: egy `x` és egy `y` nevű egész számból.

A típus alapján már létrehozhatunk változót:

```
struct Pont p;
```

A `p` változó tehát egy pontot tárol. Az `x` és `y` koordinátát a pontoperátorral érhetjük el:

```
p.x = 3;
p.y = 5;
```

A teljes program:

```
#include <stdio.h>

struct Pont {
    int x;
    int y;
};

int main() {
    struct Pont p;

    p.x = 3;
    p.y = 5;

    printf("A pont koordinatai: (%i, %i)\n", p.x, p.y);

    return 0;
}
```

A `p.x` jelentése: a `p` struktúraváltozó `x` mezője. Hasonlóan, a `p.y` a `p` változó `y` mezőjét jelenti.

14.2. Mezők beolvasása és kiírása

A struktúra mezői ugyanúgy használhatók, mint más változók. Ha egy mező értékét be szeretnénk olvasni, akkor ugyanazokat a szabályokat alkalmazzuk, mint korábban.

```
#include <stdio.h>

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

```
int main() {
    struct Pont p;

    scanf("%i %i", &p.x, &p.y);

    printf("(%i, %i)\n", p.x, p.y);

    return 0;
}
```

Figyeljük meg, hogy a `scanf` hívásban `&p.x` és `&p.y` szerepel. Azért van szükség címoperátorra, mert egész számokat olvasunk be, és a `scanf`-nek tudnia kell, hová írja a beolvasott értékeket.

A pontoperátor erősebben köt, mint a címoperátor, ezért az `&p.x` jelentése: a `p` struktúra `x` mezőjének címe.

14.3. Struktúra különböző típusú mezőkkel

A struktúrák egyik nagy előnye, hogy különböző típusú adatokat is összefoghatnak. Nézzünk egy tanulót, akinek neve és pontszáma van.

```
struct Tanulo {
    char nev[50];
    int pontszam;
};
```

A `nev` mező karaktertömb, vagyis karakterlánc tárolására alkalmas. A `pontszam` mező egész szám.

```
#include <stdio.h>

struct Tanulo {
    char nev[50];
    int pontszam;
};

int main() {
    struct Tanulo t;

    scanf("%49s", t.nev);
    scanf("%i", &t.pontszam);

    printf("%s pontszama: %i\n", t.nev, t.pontszam);
```

```
    return 0;
}
```

A `scanf("%49s", t.nev);` hívásban nincs `&` jel a `t.nev` előtt. Ennek oka ugyanaz, mint a karakterláncoknál korábban: a karaktertömb neve sok helyzetben az első elem címévé alakul. A `pontszam` mező viszont egyszerű egész változó, ezért ott szükség van a címoperátorra:

```
scanf("%i", &t.pontszam);
```

[i] Megjegyzés. Ha a név szóközt is tartalmazhat, például Kovacs Anna, akkor a `%s` nem elegendő, mert csak az első üres karakterig olvas. Ilyenkor használhatjuk az előző fejezetben korábban látott scanset formát:

```
scanf(" %49[^\n]", t.nev);
```

Ez sorvégig olvas, legfeljebb 49 karaktert, és helyet hagy a lezáró `'\0'` karakternek.

14.4. Struktúrák tömbje

Ha több tanuló adatait szeretnénk tárolni, akkor struktúrákból álló tömböt használhatunk.

```
struct Tanulo osztaly[30];
```

Ez a tömb 30 darab `struct Tanulo` típusú elemet tartalmaz. Az `osztaly[0]` az első tanuló, az `osztaly[1]` a második, és így tovább.

Az egyes tanulók mezőit így érjük el:

```
osztaly[i].nev
osztaly[i].pontszam
```

Példa: olvassuk be `n` tanuló nevét és pontszámát, majd írjuk ki a legjobb pontszámot elért tanuló nevét.

```
#include <stdio.h>

#define MAXN 30
```

```
struct Tanulo {
    char nev[50];
    int pontszam;
};

int main() {
    int n;
    struct Tanulo osztaly[MAXN];

    scanf("%i", &n);

    for (int i = 0; i < n; i++) {
        scanf("%49s", osztaly[i].nev);
        scanf("%i", &osztaly[i].pontszam);
    }

    int legjobb = 0;

    for (int i = 1; i < n; i++) {
        if (osztaly[i].pontszam > osztaly[legjobb].pontszam) {
            legjobb = i;
        }
    }

    printf("A legjobb tanulo: %s\n", osztaly[legjobb].nev);
    printf("Pontszama: %i\n", osztaly[legjobb].pontszam);

    return 0;
}
```

A `legjobb` változó nem magát a legjobb tanulót tárolja, hanem annak indexét a tömbben. Ez a korábbi minimum- és maximumkereséshez hasonló gondolat: nemcsak a legnagyobb értéket kereshetjük meg, hanem azt is, hogy hol található.

14.5. Struktúra átadása függvénynek

Struktúraváltozót függvénynek is átadhatunk. Először nézzünk egy olyan függvényt, amely kiír egy pontot.

```
void pont_kiir(struct Pont p) {
    printf("(%i, %i)\n", p.x, p.y);
}
```

A teljes program:

```
#include <stdio.h>

struct Pont {
    int x;
    int y;
};

void pont_kiir(struct Pont p);

int main() {
    struct Pont p;

    p.x = 2;
    p.y = 7;

    pont_kiir(p);

    return 0;
}

void pont_kiir(struct Pont p) {
    printf("(%i, %i)\n", p.x, p.y);
}
```

Itt a `p` struktúra érték szerint adódik át a függvénynek. Ez azt jelenti, hogy a függvény saját példányt kap belőle. Ha a függvény módosítaná a saját `p` paraméterét, az nem változtatná meg a `main` függvényben lévő eredeti pontot.

Nézzünk egy példát:

```
void pont_nullaz(struct Pont p) {
    p.x = 0;
    p.y = 0;
}
```

Ez a függvény csak a másolatot nullázná, az eredeti pontot nem.

Ha azt szeretnénk, hogy a függvény módosítsa az eredeti struktúrát, akkor a struktúra címét kell átadnunk.

```
void pont_nullaz(struct Pont *p) {
    (*p).x = 0;
    (*p).y = 0;
}
```

A hívás:

```
pont_nullaz(&p);
```

A `(*p).x` azt jelenti, hogy előbb eljutunk ahhoz a struktúrához, amelyre `p` mutat, majd annak vesszük az `x` mezőjét.

Mivel ez gyakori művelet, a C nyelv rövidebb írásmódot is ad rá:

```
p->x = 0;
p->y = 0;
```

A `p->x` tehát ugyanazt jelenti, mint az `(*p).x`.

A teljes program:

```
#include <stdio.h>

struct Pont {
    int x;
    int y;
};

void pont_nullaz(struct Pont *p);

int main() {
    struct Pont p;

    p.x = 2;
    p.y = 7;

    pont_nullaz(&p);

    printf("(%i, %i)\n", p.x, p.y);

    return 0;
}

void pont_nullaz(struct Pont *p) {
    p->x = 0;
    p->y = 0;
}
```

[i] Megjegyzés. Ha egy függvénynek csak olvasnia kell a struktúra mezőit, átadhatjuk érték szerint. Ha viszont módosítania kell az eredeti struktúrát, akkor

mutatót adunk át. Nagyobb struktúráknál akkor is gyakori a mutatóval történő átadás, ha nem módosítjuk az adatot, mert így nem kell az egész struktúrát lemásolni.

14.6. Struktúrák és `typedef`

A `struct Tanulo` típus neve minden változódefiníciónál kiírható:

```
struct Tanulo t;
struct Tanulo osztaly[30];
```

Ez teljesen helyes, de kissé hosszú. A `typedef` segítségével rövidebb típusnevet is adhatunk.

```
typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;
```

Ezután a típus neve egyszerűen `Tanulo` lesz:

```
Tanulo t;
Tanulo osztaly[30];
```

Példa:

```
#include <stdio.h>

typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;

int main() {
    Tanulo t;

    scanf("%49s", t.nev);
    scanf("%i", &t.pontszam);

    printf("%s: %i\n", t.nev, t.pontszam);

    return 0;
}
```

A `typedef` nem hoz létre új adatot, csak egy típusnak ad rövidebb, kényelmesebb nevet.

[i] Megjegyzés. A `struct Tanulo { ... };` és a `typedef struct { ... } Tanulo;` forma között kezdő szinten az a legfontosabb különbség, hogy az első esetben a típus teljes neve `struct Tanulo`, a második esetben pedig egyszerűen `Tanulo`.

14.7. Összetartozó adatok egyben

A struktúrák segítségével saját összetett típusokat hozhatunk létre. Ez új gondolkodásmódot jelent: már nemcsak különálló változókban gondolkodunk, hanem olyan egységekben, amelyek több adatot fognak össze.

Sőt, a struktúrák akár egymásba is ágyazhatók. Például ha már definiáltunk egy dátumot leíró típust, akkor ezt felhasználhatjuk például egy könyv adatainak leírásakor:

```
typedef struct {
    int ev;
    int honap;
    int nap;
} Datum;

typedef struct {
    char cim[100];
    char szerzo[100];
    Datum kiadas_datuma;
    int oldalszam;
} Konyv;
```

Itt a `Konyv` struktúra egyik mezője maga is struktúra típusú: a `kiadas_datuma` mező `Datum` típusú adat.

A mezők elérése ilyenkor egymás után írt pontoperátorokkal történik:

```
Konyv k;

k.kiadas_datuma.ev = 2024;
k.kiadas_datuma.honap = 5;
k.kiadas_datuma.nap = 12;
```

A `k.kiadas_datuma.ev` kifejezés jelentése: a `k` könyv `kiadas_datuma` mezőjének `ev` mezője.

Ez a megoldás jól mutatja, hogy a struktúrák segítségével összetettebb adatmodelleket is felépíthetünk. Nem kell minden adatot külön-külön mezőként felsorolnunk, hanem az összetartozó részeket önálló struktúrába szervezhetjük, majd ezeket nagyobb struktúrák részeként használhatjuk.

A lényeg mindig ugyanaz: ha több adat logikailag ugyanahhoz az objektumhoz tartozik, akkor érdemes őket egy struktúrába szervezni. Ily módon a struktúrák hidat képeznek az egyszerű változók és az összetettebb adatmodellek között. Segítségükkel a programjaink közelebb kerülnek ahhoz, ahogyan a valóságban is gondolkodunk: pontokról, tanulókról, könyvekről, versenyzőkről, dátumokról és más összetett objektumokról.

Gyakorló feladatok

Definiálj `Pont` nevű struktúrát, amely egy síkbeli pont két koordinátáját tárolja.

- Írj függvényt, amely kiszámítja két pont Manhattan-távolságát. Két pont Manhattan-távolsága az `x` koordináták különbségének abszolút értéke plusz az `y` koordináták különbségének abszolút értéke.

```
int manhattan_tavolsag(Pont a, Pont b);
```

Definiálj `Tanulo` nevű struktúrát, amely egy tanuló nevét és pontszámát tárolja.

- Adott egy `Tanulo` elemekből álló tömb. Írj függvényt, amely kiszámítja a tanulók átlagpontszámát.

```
double atlag_pontszam(Tanulo *t, int n);
```

- Adott egy `Tanulo` elemekből álló tömb. Írj függvényt, amely név alapján megkeres egy tanulót. Ha a keresett név nem szerepel a tömbben, térjen vissza `-1` értékkel.

```
int tanulo_keres(Tanulo *t, int n, char nev[]);
```

- Írj függvényt, amely pontszám szerint csökkenő sorrendbe rendezi a tanulókat. Használd valamelyik, korábbi fejezetben bemutatott rendezési algoritmust.

```
void pontszam_szerint_rendez(Tanulo *, int n);
```

Definiálj `Datum` nevű struktúrát, majd definiálj `Konyv` nevű struktúrát, amely a könyv címét, szerzőjét, kiadási dátumát és oldalszámát tárolja.

- Írj függvényt, amely kiírja egy könyv adatait.

```
void konyv_kiir(Konyv k);
```

- Adott egy `Konyv` elemekből álló tömb. Írj függvényt, amely megadja a legrégebben kiadott könyv indexét.

```
int legregebbi_konyv(Konyv *k, int n);
```

[i] Megjegyzés. A struktúratömbök esetén különösen indokolt lehet a dinamikus memóriafoglalás használata. Egy struktúra több mezőből állhat, és akár nagyobb karaktertömböket is tartalmazhat. Emiatt egy sok elemből álló struktúratömb jelentős memóriaterületet foglalhat. Ha a tömb mérete csak futás közben derül ki, akkor a `main` függvényben a `malloc`, `calloc` és `free` függvényekkel foglalhatunk, illetve szabadíthatunk fel memóriát, ahogy azt a 8. fejezetben bemutattuk.

Ezt sugalltuk azzal is, hogy a gyakorló feladatok függvénydeklarációiban a struktúratömböket mutatóként adtuk át. A `Tanulo *t` paraméter, ahogy a `Tanulo t[]` alak is, jelenthet hagyományos tömböt is, de jelenthet dinamikusan lefoglalt struktúratömböt is. A függvényen belül mindkét esetben ugyanúgy használhatjuk az indexelést: `t[i].pontszam, t[i].nev`.



AI-csepp

Kérdezd meg az AI-t!

- Melyek a tipikus hibák a struktúrák használatakor?
- Miben hasonlít és miben különbözik a `struct` típus az egyszerű típusoktól?
- Miben hasonlít és miben különbözik a `struct` típus a tömbtípustól?
- Általánosabban mire használható a `typedef`? Milyen más alkalmazásai lehetnek a `struct` típusok rövidebb elnevezésén kívül?

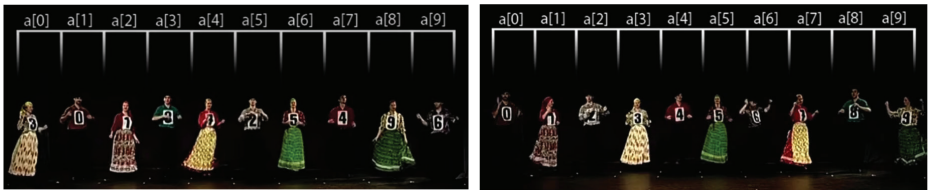
14.8. AlgoRythmics szál: kiválasztásos rendezés struktúrákra



▦ **Algoritmikus mintázat.** A kiválasztásos rendezés lényege egyszerűen: minden lépésben megkeressük a még rendezetlen rész legkisebb elemét, majd kicseréljük a rendezetlen rész első elemével. Így balról jobbra haladva egyre nő a rendezett rész.

Az AlgoRythmics-videóban az algoritmus táncos szemléltetése kissé másként dolgozik (14.1. ábra). A rendezetlen rész első pozíciójában álló elemhez hasonlítják a soron következő elemeket, és ha a vizsgált elem kisebb nála, akkor helyet

cserélnek. Így a rendezetlen rész legkisebb eleme cserék sorozatán keresztül kerül a rendezetlen rész első pozíciójába.



14.1. ábra. *Pillanatképek a kiválasztásos rendezést cigány néptánccal ábrázoló AlgoRythmics videóból*

A gyakorlatban általában másként valósítjuk meg a kurrens rendezetlen szakasz legkisebb elem előrehozását: a bejárás során csak megjegyezzük a legkisebb vagy legnagyobb elem indexét, majd a keresés végén egyetlen cserével tesszük az aktuális helyére. Struktúrák esetén ez különösen hasznos, hiszen egy csere nemcsak egyetlen számot mozgat meg, hanem a teljes struktúrát, például a tanuló nevét és pontszámát együtt.

Tegyük fel, hogy van egy struktúratömbünk, amely tanulók nevét és pontszámát tárolja:

```
typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;
```

A feladat az, hogy a tanulókat pontszám szerint csökkenő sorrendbe rendezzük. Vagyis az első helyre a legnagyobb pontszámú tanuló kerüljön, utána a második legjobb, és így tovább.

A kiválasztásos rendezés gondolata most így hangzik:

```
minden pozícióra:
    keresd meg a még rendezetlen rész legnagyobb pontszámú
tanulóját
    cseréld ki az aktuális pozíción lévő tanulóval
```

A különbség a korábbi rendezésekhez képest az, hogy itt nem egyetlen egész számot cserélünk, hanem egy teljes struktúrát. A csere tehát a névvel és a pontszámmal együtt mozgatja a tanulót.

```
void csere(Tanulo *a, Tanulo *b) {
```

```
    Tanulo seged = *a;
    *a = *b;
    *b = seged;
}
```

Ezután a rendezés:

```
void pontszam_szerint_rendez(Tanulo *t, int n) {
    for (int i = 0; i < n - 1; i++) {
        int legjobb = i;

        for (int j = i + 1; j < n; j++) {
            if (t[j].pontszam > t[legjobb].pontszam) {
                legjobb = j;
            }
        }

        csere(&t[i], &t[legjobb]);
    }
}
```

A `t[j].pontszam > t[legjobb].pontszam` feltétel azt vizsgálja, hogy a `j` indexű tanuló pontszáma nagyobb-e az eddig legjobbnak talált tanuló pontszámánál. A legjobb változó tehát nem magát a tanulót tárolja, hanem annak indexét a tömbben.

Figyeljük meg a cserehívást is:

```
csere(&t[i], &t[legjobb]);
```

Itt két `Tanulo` struktúra címét adjuk át a csere függvénynek. A függvény ezeken a címeken keresztül cseréli ki a két teljes struktúrát.

A teljes struktúra másolása azért működik, mert C-ben a struktúraváltozók értékadással másolhatók. Ez fontos különbség a karaktertömbökhöz képest. Egy karakterláncot nem másolhatunk egyszerűen `s1 = s2;` alakban, de két azonos típusú struktúraváltozó között az értékadás megengedett. Ha a struktúra karaktertömb mezőt tartalmaz, akkor az értékadás a karaktertömb mező tartalmát is átmásolja.

Ez a példa jól mutatja a struktúrák erejét: a rendezési algoritmus ugyanaz marad, de az elemek már összetett adatok. Nem külön névtömböt és pontszámtömböt rendezünk, hanem tanulókat mint egységes adatcsomagokat.



AI-csepp
Kérdezd meg az AI-t!
• Mennyi a kiválasztásos rendezés időbonyolultsága legjobb és legrosszabb esetben?
• Mennyivel hatékonyabb az itt használt megvalósítás az AlgoRythmics-videóban eltáncolt változatnál?
• A buborékos és a beszúró rendezésnek „ki lehetett nyitni a szemét”: egyetlen bejárásból felismerhették, ha a bemeneti tömb már rendezett. Miért nem valósítható meg ugyanez ilyen egyszerűen a kiválasztásos rendezés esetében?

14.9. BBC szál: ütemezés – amikor a sorrend számít



A BBC-filmben több olyan példa is szerepel, amelyben nem az a kérdés, hogy mit kell elvégezni, hanem az, hogy milyen sorrendben (14.2. ábra). Ilyen helyzetekkel a mindennapokban is gyakran találkozunk: repülőgépek felszállási sorrendje, munkák kiosztása, feladatok határidő szerinti elvégzése, betegek ellátási sorrendje vagy gépek használatának beosztása.



14.2. ábra. *A BBC filmje a Heathrow repülőtér példáján keresztül szemlélteti az ütemezési algoritmusok fontosságát*

Az ilyen feladatokat *ütemezési problémáknak* nevezzük. Ütemezésről beszélünk, amikor több teendő, kérés vagy esemény közül kell sorrendet kialakítanunk valamilyen szabály alapján. A cél sokféle lehet: minél kevesebb legyen a várakozási idő, a sürgős feladatok kerüljenek előre, ne késsünk le határidőket, vagy minél jobban kihasználjunk egy korlátozott erőforrást.

Ebben a fejezetben egy egyszerűsített ütemezési feladatot vizsgálunk. Tegyük fel, hogy egy rendelőben betegeket kell sorba állítanunk. Minden beteghez tartozik egy név, egy érkezési idő és egy sürgősségi szint.

```
typedef struct {
    char nev[50];
    int erkezes;
    int surgosseg;
} Beteg;
```

Az `erkezes` mező azt jelzi, hogy a beteg hányadik percben érkezett. A `surgosseg` mező azt mutatja, mennyire sürgős az eset: minél nagyobb ez az érték, annál előbb kellene sorra kerülnie.

Most használjunk egy egyszerű szabályt: előbb kerüljön sorra a sürgősebb beteg; azonos sürgősség esetén az kerüljön előrébb, aki korábban érkezett.

Mindez rendezést feltételez. A rendezésről már láttuk, hogy általában két alapműveletre épül: az elemek összehasonlítására és – ha szükséges – a cseréjükre. Mivel itt a sorrendről nem egyetlen mező, hanem két feltétel alapján döntünk, az összehasonlítást is külön függvénybe szervezzük, nemcsak a cserét. Ez a megközelítés azért is hasznos, mert elkülöníti a rendezési stratégiát a két alapművelettől.

Az `elobb_kerul_sorra_e` függvényben először a `surgosseg` mezőt vizsgáljuk, és csak azonos sürgősség esetén vesszük figyelembe az érkezési időt az `erkezes` mező alapján.

A `beteg_csere` függvény két teljes `Beteg` struktúrát cserél ki. A csere során a név, az érkezési idő és a sürgősségi szint együtt mozog, vagyis a beteg adatai nem esnek szét.

```
int elobb_kerul_sorra_e(Beteg a, Beteg b) {
    if (a.surgosseg > b.surgosseg) {
        return 1;
    }

    if (a.surgosseg < b.surgosseg) {
        return 0;
    }

    return a.erkezes < b.erkezes;
}

void beteg_csere(Beteg *a, Beteg *b) {
    Beteg seged = *a;
    *a = *b;
    *b = seged;
}
```

Az alább következő `utemez1`, `utemez2` és `utemez3` függvény ugyanazt az ütemezési szabályt valósítja meg, de három különböző rendezési mintát használ: rendre buborékos, beszúró és kiválasztásos rendezést.

```
void utemez1(Beteg betegek[], int n) {
    for (int i = n - 1; i > 0; i--) {
        for (int j = 0; j < i; j++) {
            if (elobb_kerul_sorra_e(betegek[j + 1], betegek[j]))
{
                beteg_csere(&betegek[j], &betegek[j + 1]);
            }
        }
    }
}

void utemez2(Beteg betegek[], int n) {
    for (int i = 1; i < n; i++) {
        Beteg aktualis = betegek[i];
        int j = i - 1;

        while (j >= 0 && elobb_kerul_sorra_e(aktualis,
betegek[j])) {
            betegek[j + 1] = betegek[j];
            j--;
        }

        betegek[j + 1] = aktualis;
    }
}

void utemez3(Beteg betegek[], int n) {
    for (int i = 0; i < n - 1; i++) {
        int kovetkezo = i;

        for (int j = i + 1; j < n; j++) {
            if (elobb_kerul_sorra_e(betegek[j],
betegek[kovetkezo])) {
                kovetkezo = j;
            }
        }

        beteg_csere(&betegek[i], &betegek[kovetkezo]);
    }
}
```

A rendezőfüggvény így három egymástól jól elkülöníthető gondolatot kapcsol össze:

```
mi alapján döntjük el két elem sorrendjét?   -> összehasonlító
                                                függvény
hogyan cserélünk ki két elemet?              -> cserefüggvény
milyen rendezési mintát használunk?          -> rendezési
                                                stratégia
```

Ez a szétválasztás azért hasznos, mert ugyanaz a rendezési stratégia többféle adattípusra és többféle rendezési szempontra is alkalmazható. A rendezés általános mintája nem változik attól, hogy számokat, tanulókat, betegeket vagy repülőgépeket rendezünk. Ami változik, az az összehasonlítás szabálya és esetleg az elemek cseréjének módja.

Ez fontos lépés a programozói gondolkodásban: ha más adatokkal dolgozunk, vagy más sorrendet szeretnénk, sokszor elég az összehasonlító függvényt módosítani, miközben maga a rendezési minta változatlan marad. Ezzel előkészítettük a terepet az alábbiakban bemutatott `qsort` könyvtári függvényhez.

[i] Megjegyzés. Valódi ütemezési feladatokban sokkal több szempont is megjelenhet: a vizsgálat várható hossza, a rendelkezésre álló orvosok száma, a határidők, a felszerelések elérhetősége vagy az igazságosság kérdése. Itt egyszerűsített modellt használtunk, mert most a fő cél az volt, hogy lássuk: egy ütemezési feladatban a sorrendet gyakran több mező alapján határozzuk meg.

[i] Megjegyzés. A fenti megoldás egy egyszerű prioritási szabály alapján állítja sorba a betegeket: előbb a sürgősebb esetek kerülnek sorra, azonos sürgősség esetén pedig az, aki korábban érkezett. Ez rokon a mohó algoritmusok gondolatával, mert minden összehasonlításnál a pillanatnyilag előnyösebbnek tekintett beteget választjuk előbbre. Valódi optimalizálási feladatnál azonban pontosan meg kellene mondanunk, mit szeretnénk minimalizálni vagy maximalizálni, például az átlagos várakozási időt, a leghosszabb várakozást vagy az összvárakozási időt.



AI-csepp

Kérd meg az AI-t!

- Magyarázd el a mohó módszer lényegét.
- Adj példát optimalizálási mohó ütemezési problémára.

14.10. ECN szál: hatékony rendezés és keresés könyvtári függvényekkel



Az ECN-versenyeken évről évre előkerülnek olyan feladatok, amelyekben sok adatot kell feldolgozni. Ilyenkor az adatok gyakran nem egyszerű számok, hanem összetett rekordok: azonosítók, nevek, kódok, pontszámok, időpontok vagy más mezők tartoznak össze. Ezek tárolására természetes eszköz a `struct`.

A nagy bemenet miatt azonban nem mindegy, hogyan dolgozunk az adatokkal. Ha egy struktúratömböt sokszor kell keresni, csoportosítani vagy feldolgozni, gyakran érdemes előbb rendezni. A rendezés célja kétféle lehet: vagy az, hogy később gyors keresést végezhessünk, vagy az, hogy az összetartozó elemek egymás mellé kerüljenek, és így egyszerűbben feldolgozhatók legyenek.

C-ben ehhez a szabványos könyvtár két fontos függvényt is kínál: `qsort`, `bsearch`.

A `qsort` hatékony általános rendezőfüggvény. Nemcsak egész számokat tud rendezni, hanem bármilyen típusú elemekből álló tömböt, így struktúratömböt is. Ehhez meg kell adnunk, hol kezdődik a tömb, hány eleme van, mekkora egy elem mérete, és milyen összehasonlító függvény alapján történjen a rendezés.

A `bsearch` rendezett tömbben keres a bináris keresés stratégiája alapján. Ez azt jelenti, hogy csak akkor használható helyesen, ha a tömb már ugyanazon szempont szerint rendezett, amely szerint keresni szeretnénk. A `bsearch`-nek van egy további paramétere is, amelyet elsőként kell megadnunk: a keresett elem címe.

Betét: A `qsort` azért tud általános rendezőfüggvény lenni, mert nem akarja tudni, milyen típusú elemeket rendez. Úgy tekint a tömbre, mintha az fekete dobozok sorozata lenne. Az első három paraméter alapján a `qsort` tudja, hol kezdődik a tömb, hány doboz van benne, és mekkora egy doboz mérete.

A rendezésnek két alapművelete van: az összehasonlítás és a csere. A cserét a `qsort` fekete dobozként is el tudja végezni: nem kell tudnia, mi van a dobozok belsejében, elegendő, ha pontosan ismeri a dobozok helyét és méretét.

Az összehasonlítás viszont nem végezhető el fekete dobozként. Ezért az összehasonlító függvényt a programozó írja meg, mert ő tudja, mi van a dobozokban. Mivel ezt a függvényt az általános `qsort` hívja meg, a két összehasonlítandó elem címét `const void *` típusú mutatóként kapjuk meg, ami itt azt jelenti: „valamilyen típusú adat címe”.

Programozóként az összehasonlító függvényben először vissza kell alakítanunk a mutatókat olyan típusúvá, amilyen elemeket valójában rendezünk, majd meg kell írnunk az összehasonlítás szabályát. Az összehasonlító függvény visszatérési értékének jelentése hasonló kell legyen ahhoz, amit a 12. fejezetben a `strcmp` függvénynél láttunk. Ha az `int` típusú visszatérési érték negatív, akkor az

első elemnek kell előbb állnia a rendezett sorrendben; ha nulla, akkor a két elem a rendezés szempontjából egyenértékű; ha pedig pozitív, akkor az első elemnek kell később állnia.

Bár az összehasonlító függvényt nekünk kell megírnunk, ez általában rövid és egyszerű. A nagy segítség az, hogy magát a rendezési stratégiát nem kell leprogramoznunk. A `qsort` hatékony könyvtári rendezőfüggvény; nekünk csak azt kell megmondanunk, hogy két elem közül melyik álljon előbb.

A 15. mesterfejezetben visszatérünk arra az érdekes jelenségre, hogy függvénynek paraméterként függvényt is átadhatunk.

A 2023-as ECN-feladatban (*Problem J: Find the Identical Twins and Triplets*) személyek adatait kellett feldolgozni. Egy személyhez több adat tartozott: személyi kód, DNS-aláírás, örökbefogadási jel és név. Ezeket természetesen egy struktúrában tárolhatjuk:

```
typedef struct {
    int p_code;
    char dna[12];
    char adopted;
    char name[28];
} Person;
```

A személyek tömbjét dinamikusan foglaljuk le, hiszen a bemenet mérete nagy lehet:

```
Person *arr;

arr = malloc(n * sizeof(Person));
```

Ezután a tömb rendezhető például személyi kód szerint:

```
qsort(arr, n, sizeof(Person), person_cmp);
```

Az összehasonlító függvény egy lehetséges alakja:

```
int person_cmp(const void *p1, const void *p2) {
    const Person *a = p1;
    const Person *b = p2;

    if (a->p_code < b->p_code) {
        return -1;
    }
```

```
    if (a->p_code > b->p_code) {
        return 1;
    }

    return 0;
}
```

A rendezés után a tömbben a személyek növekvő személyi kód szerint állnak. Ez azért hasznos, mert a feladat feltétele szerint az azonos DNS-ű ikrek és hármasikrek a személyi kód szerinti rendezés után egymás mellé kerülnek. Így a megoldásnak nem kell minden személyt minden másikkal összehasonlítania. Elég balról jobbra bejárni a rendezett tömböt, és az egymás melletti elemeket vizsgálni.

A feladat azt kéri, hogy írjuk ki azokat az iker- és hármasiker csoportokat, amelyekben legalább egy személy örökbefogadott. Ennek kiolvasását a rendezett tömbből az olvasóra hagyjuk.

Ez az ECN-feladat jól mutatja, miért fontos a hatékony rendezés. A rendezés nem öncélú lépés: előkészíti az adatfeldolgozást. Az összetartozó elemek egymás mellé kerülnek, és így a további feldolgozás egyszerűbbé és gyorsabbá válik.



AI-csepp

Kérdezd meg az AI-t!

- Egy `qsort`-hoz írt összehasonlító függvényben, egész számok összehasonlításakor, működhet a `return a->p_code - b->p_code;` alak. Milyen gond lehet ezzel?
- Miért nem jó ötlet valós számok összehasonlításakor a `return a->ertek - b->ertek;` alakot használni egy `qsort`-hoz írt összehasonlító függvényben?
- Hogyan néznek ki a `qsort`, a `bsearch` és az általuk használt összehasonlító függvények deklarációi? Mit jelentenek az egyes paraméterek?
- Mit jelent a const a `qsort`-hoz és `bsearch`-hez írt összehasonlító függvények paramétereiben, például a `const void *p1` és `const void *p2` esetén?
- Egy hatékony rendezés, amelynek időbonyolultsága `O(n log n)`, költségesebb, mint egyetlen lineáris keresés, amelynek időbonyolultsága `O(n)`. Mikor éri meg mégis rendezni egy adatsort azért, hogy utána hatékonyan, bináris kereséssel, `O(log n)` időben tudjunk benne keresni?

14.11. Hiúz szál: házasságok és válások nyilvántartása



A hiúzrezervátumban minden hiúzt egyedi azonosítóval tartanak nyilván. A nyilvántartásban szerepel, hogy az adott egyed hím vagy nőstény. Kezdetben egyik hiúznak sincs párja.

A rezervátum munkatársai ezután sok kérést dolgoznak fel. Egy kérés lehet „házasságkötési” kérés vagy „válási” kérés. A program feladata, hogy a kéréseket sorban feldolgozza, megszámolja az érvényes „házasságkötéseket” és „válásokat”, valamint az egyes hibakódokat.

A hibakódok jelentése legyen a következő:

```
1. hibakód: az első azonosító nem hímhez tartozik
2. hibakód: a második azonosító nem nőstényhez tartozik
3. hibakód: "házasságkötésnél" a hímnek már van párja
4. hibakód: "házasságkötésnél" a nősténynek már van párja
5. hibakód: "válásnál" a két hiúz nem egymás párja
```

A szabályokat mindig ebben a sorrendben vizsgáljuk. Ez azért fontos, mert egy kérés akár több szempontból is hibás lehetne, de ilyenkor csak az elsőként megtalált hibát számoljuk.

A feladat azért illik ehhez a fejezethez, mert egy hiúz adatai természetesen összetartoznak: azonosító, pár azonosítója, aktuális állapot. Ezeket struktúrában tárolhatjuk.

```
typedef struct {
    int id;
    int par;
    int foglalt;
} Hiuz;
```

A nyilvántartásban sok hiúz és sok kérés szerepelhet, ezért nem lenne hatékony minden kérésnél lineárisan végigkeresni a teljes adatállományt. Ehelyett előkészítjük az adatokat: külön tömbben tároljuk a hímeket és külön tömbben a nőstényeket, majd mindkét tömböt azonosító szerint rendezzük.

```
Hiuz *himek = calloc(n, sizeof(Hiuz));
Hiuz *nostenyek = calloc(n, sizeof(Hiuz));
```

Ez a tárolási mód azért előnyös, mert egy „házasságkötési” vagy „válási” kérésben az első azonosítónak hímhez, a másodiknak nőstényhez kell tartoznia. Ezért az első azonosítót mindig a hímek között, a másodikat mindig a nőstények

között keressük. Ha az első keresés sikertelen, akkor az 1-es hibakódot kapjuk; ha a második keresés sikertelen, akkor a 2-es hibakódot.

A feldolgozás menete:

```
1. Olvasd be a hiúzok adatait.
2. A hímeket és a nőstényeket külön struktúratömbben tárold.
3. Rendezd mindkét tömböt azonosító szerint qsort-tal.
4. Minden kérésnél keresd meg az érintett hímet és nőstényt
   bsearch segítségével.
5. Ellenőrizd a kérés szabályosságát a hibakódok sorrendjében.
6. Érvényes kérés esetén módosítsd mindkét érintett hiúz adatait.
7. Számold meg az érvényes kéréseket és a hibakódokat.
```

Amennyiben a beolvasás és a szétválogatás nyomán `nh` adat kerül a `himek` tömbbe, `nn` adat pedig a `nostenyek` tömbbe, a két `qsort`-hívás az alábbi lehet:

```
qsort(himek, nh, sizeof(Hiuz), hiuz_cmp);
qsort(nostenyek, nn, sizeof(Hiuz), hiuz_cmp);
```

A `hiuz_cmp` összehasonlító függvény megírását az olvasóra hagyjuk.

A `bsearch` használatához egy keresett azonosítóból ideiglenes `Hiuz` struktúrát készítünk, majd ezt keressük a megfelelő tömbben.

```
Hiuz kulcs;
Hiuz *him;
Hiuz *nosteny;

kulcs.id = id1;
him = bsearch(&kulcs, himek, nh, sizeof(Hiuz), hiuz_cmp);

kulcs.id = id2;
nosteny = bsearch(&kulcs, nostenyek, nn, sizeof(Hiuz), hiuz_cmp);
```

Figyeljük meg, hogy a keresés eredménye mutató. Ha a `bsearch` nem talál megfelelő elemet, akkor `NULL` értékkel tér vissza. Ha viszont találat van, akkor közvetlenül a tömbben lévő elem címét kapjuk meg. Ez azt jelenti, hogy a megtalált hiúz adatait módosítani is tudjuk.

„Házasságkötési” kérés esetén először ellenőrizzük, hogy az első azonosító hímhez, a második nőstényhez tartozik-e. Ezután azt vizsgáljuk, hogy valamelyiküknek van-e már párja.

```
if (him == NULL) {
```

```
        hiba[1]++;
    } else if (nosteny == NULL) {
        hiba[2]++;
    } else if (him->foglalt) {
        hiba[3]++;
    } else if (nosteny->foglalt) {
        hiba[4]++;
    } else {
        him->foglalt = 1;
        nosteny->foglalt = 1;

        him->par = nosteny->id;
        nosteny->par = him->id;

        ervenyes_hazassag++;
    }
```

„Válási” kérés esetén szintén először a nemekre vonatkozó feltételeket vizsgáljuk. Ha ezek rendben vannak, akkor ellenőrizzük, hogy a két hiúz valóban egymás párja-e.

```
if (him == NULL) {
    hiba[1]++;
} else if (nosteny == NULL) {
    hiba[2]++;
} else if (him->par != nosteny->id || nosteny->par != him->id) {
    hiba[5]++;
} else {
    him->foglalt = 0;
    nosteny->foglalt = 0;

    him->par = 0;
    nosteny->par = 0;

    ervenyes_valas++;
}
```

[i] Megjegyzés. Remélhetőleg ennyi információ már elegendő ahhoz, hogy az olvasó össze tudjon rakni belőle egy működőképes programot. Azzal, hogy a kiegészítést az olvasóra hagyjuk, az is célunk, hogy előmozdítsuk a megírt részletek kódolvasását. Ez fontos programozói képesség: nemcsak kódot írni kell tudnunk, hanem mások kódját– vagy akár saját korábbi kódunkat – is meg kell értenünk.

Ez a feladat jól mutatja, mikor éri meg rendezni egy adatsort. Egyetlen keresés kedvéért nem feltétlenül lenne hasznos rendezni. Ha azonban nagyon sok kérésre kell válaszolnunk, akkor a rendezés előfeldolgozásként megtérül: utána minden keresés gyorsan elvégezhető bináris kereséssel.

A feladat másik tanulsága az adatszervezés fontossága. A hímeket és a nőstényeket tárolhatnánk egyetlen közös tömbben is, külön `nem` mezővel. Most azonban külön tömböket használunk, mert a kérések feldolgozásakor az első azonosítót mindig a hímek között, a másodikat mindig a nőstények között kell keresni. Így a tárolási mód közvetlenül támogatja a későbbi műveleteket.



AI-csepp

Kérdezd meg az AI-t!

- Mi változna a megoldásban, ha minden hiúzt egyetlen közös struktúratömbben tárolnánk?
- Mit nyerünk azzal, hogy a hímeket és a nőstényeket külön tömbben tároljuk?
- Melyek a tipikus hibák a `qsort` és a `bsearch` használatakor?

►► Zárógondolat

Ebben a fejezetben megtanultuk, hogyan foghatunk össze több, logikailag összetartozó adatot egyetlen egységbe a `struct` segítségével. Egy pont koordinátái, egy tanuló neve és pontszáma, egy beteg érkezési ideje és állapotának súlyossága, vagy egy hiúz azonosítója és párkapcsolati állapota már nem szétszórt változókként jelent meg, hanem egy-egy összetartozó adatcsomagként.

Láttuk, hogy struktúrákból tömböt is készíthetünk, így sok hasonló objektumot kezelhetünk egységesen. Függvényeknek átadhatunk teljes struktúrákat, de mutatókon keresztül módosíthatjuk is az eredeti adatokat. A pontoperátor és a nyíloperátor segítségével elérhetjük a mezőket, a `typedef` pedig kényelmesebb típusneveket adhat a saját adatainknak.

A fejezet példái azt is megmutatták, hogy az adatszervezés legalább olyan fontos, mint maga az algoritmus. Nem mindegy, hogy minden hiúzt egy közös tömbben tárolunk-e, vagy különválasztjuk a hímeket és a nőstényeket; nem mindegy, hogy rendezetlen adatokban keresünk-e újra és újra, vagy előbb rendezzük a tömböt, majd hatékony keresést használunk. A `qsort` és a `bsearch` bevezetése arra is rávilágított, hogy a C nyelvben a könyvtári függvények sokszor általános eszközöket adnak a kezünkbe: nekünk kell megmondanunk, mit jelent két elem sorrendje.

A `struct` tehát nem csupán új szintaktikai elem. Segítségével közelebb kerülünk ahhoz, ahogyan a valós feladatokról gondolkodunk: személyekről, könyvekről, repülőkről, betegekről, versenyzőkről, eseményekről vagy bármilyen más összetett objektumról. Ettől kezdve a programjaink nemcsak adatokat tárolnak, hanem a feladat világának szereplőit is képesek természetesebb formában modellezni.

15. Állománykezelés: amikor az adatok megmaradnak

Eddigi programjaink többnyire a standard bemenetről olvastak, és a standard kimenetre írtak. A `freopen` segítségével ugyan már korábban is irányítottunk bemenetet vagy kimenetet állományba, de akkor még nem kezeltük az állományokat önálló objektumként.

Most egy lépéssel tovább megyünk. Megismerkedünk a `FILE *` típussal, az állományok megnyitásával és lezárásával, valamint a szöveges és bináris állományok közti különbséggel.

A fejezet fő gondolata egyszerű: az állománykezelés segítségével a program adatokat olvashat be háttértárról, illetve adatokat menthet el későbbi felhasználásra. Így az adatok nemcsak a program futása alatt léteznek, hanem a futás befejezése után is megmaradhatnak.



A C szálon először megnézzük, hogyan kezelhetünk állományokat önállóan, nemcsak a standard bemenet és kimenet átirányításával. A fejezet főszereplői a `FILE *`, az `fopen` és az `fclose`, valamint a szöveges állományoknál használt `fprintf` és `fscanf`. Ezután rátérünk a bináris állományokra, ahol az adatok karakteres alak helyett közvetlenül bájtokként kerülnek kiírásra és visszaolvasásra; ebben a részben a `fread` és az `fwrite` függvények kapják a főszerepet.

15.1. Állományok megnyitása és lezárása

Amikor egy program állománnyal szeretne dolgozni, először meg kell nyitnia azt. C-ben erre az `fopen` függvényt használjuk.

```
FILE *fp;

fp = fopen("adatok.txt", "r");
```

A `FILE *` típus egy állománykezelő mutató. Nem maga az állomány van benne, hanem egy olyan adatstruktúrára mutat, amelyen keresztül a C könyvtár kezelni tudja a megnyitott állományt.

Az `fopen` első paramétere az állomány neve, a második pedig a megnyitási mód.

```
"r"     olvasásra nyit meg egy létező szöveges állományt
"w"     írásra nyit meg egy szöveges állományt
"a"     hozzáfűzésre nyit meg egy szöveges állományt
```

Ha az állomány megnyitása sikertelen, az `fopen` `NULL` értéket ad vissza. Ezért a megnyitást mindig ellenőrizni kell.

```
#include <stdio.h>

int main() {
    FILE *fp;

    fp = fopen("adatok.txt", "r");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    fclose(fp);

    return 0;
}
```

Az `fclose` lezárja a megnyitott állományt. Ez azért fontos, mert íráskor az adatok egy része átmenetileg pufferben lehet, és csak később kerül ténylegesen az állományba. A lezárás gondoskodik arról, hogy az állománykezelés szabályosan fejeződjön be.

[i] Megjegyzés. A `freopen` továbbra is hasznos lehet, például a versenyprogramozásban, mert egyszerűen átirányítja a standard bemenetet vagy kimenetet. A `FILE *` használata azonban általánosabb: egyszerre több állománnyal is dolgozhatunk, és világosabban meg tudjuk mondani, melyik művelet melyik állományra vonatkozik.

15.2. Szöveges állományok

Szöveges állomány esetén az adatokat karakterek formájában tároljuk. Ha egy egész számot kiírunk egy szöveges állományba, akkor nem a memóriabeli bájtjai kerülnek ki, hanem a számjegyei.

Például a `123` egész szám szöveges állományban három karakterként jelenik meg:

```
'1' '2' '3'
```

Szöveges állományba az `fprintf` függvénnyel írhatunk. Ez hasonlít a `printf` függvényhez, de első paraméterként meg kell adnunk, melyik állományba írjon.

```
#include <stdio.h>

int main() {
    FILE *fp;

    fp = fopen("ki.txt", "w");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    fprintf(fp, "Alma\n");
    fprintf(fp, "%i\n", 123);

    fclose(fp);

    return 0;
}
```

Szöveges állományból az `fscanf` függvénnyel olvashatunk. Ez a `scanf` állományos változata.

```
#include <stdio.h>

int main() {
    FILE *fp;
    int x;

    fp = fopen("be.txt", "r");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }
```

```
    while (fscanf(fp, "%i", &x) == 1) {
        printf("%i\n", x);
    }

    fclose(fp);

    return 0;
}
```

Figyeljük meg a mintát: az `fscanf` visszatérési értékét használjuk a ciklus feltételében. A ciklus addig fut, amíg sikerül újabb egész számot beolvasni.

A 12. fejezetben látott karakteres, szavas és soronkénti beolvasási minták állományokkal is használhatók, csak a `scanf` helyett `fscanf` szerepel, és első paraméterként megadjuk az állományt.

```
char c;
while (fscanf(fp, "%c", &c) == 1) {
    /* karakter feldolgozása */
}

char szo[50];
while (fscanf(fp, "%49s", szo) == 1) {
    /* szó feldolgozása */
}

char sor[100];
while (fscanf(fp, " %99[^\n]", sor) == 1) {
    /* sor feldolgozása */
}
```

[i] Megjegyzés. Az `r`, `w` és `a` megnyitási módok szöveges állományként nyitják meg az állományt. Ez expliciten is jelezhető az `rt`, `wt` és `at` alakokkal, ahol a `t` a text szóra utal.

15.3. Szöveges és bináris állományok

A szöveges állomány ember számára olvasható. Ha megnyitjuk egy szövegszerkesztőben, számjegyeket, betűket, szóközöket és sorvégeket látunk.

A bináris állomány ezzel szemben nem feltétlenül olvasható emberi szemmel. Ilyenkor az adatokat nem karakterekké alakítva írjuk ki, hanem a memóriában lévő bájtokat mentjük el.

Például egy int típusú változó értéke a memóriában több bájton tárolódik. Szöveges állományba a számjegyeit írjuk ki, bináris állományba viszont magukat a bájtokat.

A bináris állományokat a következő módokkal nyithatjuk meg:

```
"rb"    bináris állomány megnyitása olvasásra
"wb"    bináris állomány megnyitása írásra
"ab"    bináris állomány megnyitása hozzáfűzésre
```

[i] Megjegyzés. Szöveges és bináris állomány között nem az a különbség, hogy az egyik „számokat”, a másik „szöveget” tartalmaz. Mindkettő bájtokból áll. A különbség az, hogy a szöveges állományban az adatokat karakteres formára alakítjuk, a bináris állományban viszont közvetlenebbül a memóriabeli ábrázolást mentjük.

15.4. Írás bináris állományba: `fwrite`

Bináris állományba az `fwrite` függvénnyel írhatunk. Nézzünk először egyetlen egész számot.

```
#include <stdio.h>

int main() {
    FILE *fp;
    int x = 12345;

    fp = fopen("szam.dat", "wb");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    fwrite(&x, sizeof(int), 1, fp);

    fclose(fp);

    return 0;
}
```

Az `fwrite` hívás jelentése: `fwrite(&x, sizeof(int), 1, fp);`

```
&x              honnan írunk?
sizeof(int)     mekkora egy elem?
1               hány elemet írunk?
fp              melyik állományba írunk?
```

Vagyis egy darab `int` méretű adatot írunk ki az `x` változó címéről a `fp` állományba.

15.5. Olvasás bináris állományból: `fread`

A bináris állományból az `fread` függvénnyel olvashatunk.

```
#include <stdio.h>

int main() {
    FILE *fp;
    int x;

    fp = fopen("szam.dat", "rb");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    if (fread(&x, sizeof(int), 1, fp) == 1) {
        printf("A beolvasott szam: %i\n", x);
    }

    fclose(fp);

    return 0;
}
```

Az `fread` visszatérési értéke azt mutatja meg, hány elemet sikerült beolvasni. Ezért ellenőrizzük, hogy az érték valóban `1` lett-e.

```
if (fread(&x, sizeof(int), 1, fp) == 1) {
    ...
}
```

[i] Megjegyzés. Az `fread` és `fwrite` paraméterezése nagyon hasonló. Mindkettőnél megadjuk az adatterület címét, egy elem méretét, az elemek számát és az állományt. A különbség az irány: az `fwrite` memóriából állományba ír, az `fread` állományból memóriába olvas.

15.6. Struktúra kiírása bináris állományba

A bináris állományok különösen jól kapcsolódnak a struktúrákhoz. Ha van egy `Tanulo` típusunk, akkor egy teljes `Tanulo` változót egyetlen `fwrite` hívással kiírhatunk.

```
typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;

Tanulo t1 = {"Anna", 95};

Tanulo t2;
```

Írás: `fwrite(&t1, sizeof(Tanulo), 1, fp)`
Olvasás: `fread(&t2, sizeof(Tanulo), 1, fp)`

Bináris állományok esetében nem kell külön kezelnünk a `nev` és a `pontszam` mezőt. A teljes struktúra bájtsorozata kerül az állományba, illetve ugyanilyen formában olvasható vissza.

15.7. Struktúratömb mentése és visszaolvasása

Gyakran nem egyetlen struktúrát, hanem struktúrák tömbjét szeretnénk elmenteni. Ilyenkor érdemes először az elemek számát kiírni, majd utána magát a tömböt.

```
#include <stdio.h>

typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;
```

```
int main() {
    FILE *fp;
    Tanulo t[3] = {
        {"Anna", 95},
        {"Bence", 87},
        {"Csaba", 91}
    };

    int n = 3;

    fp = fopen("tanulok.dat", "wb");

    if (fp == NULL) {
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    fwrite(&n, sizeof(int), 1, fp);
    fwrite(t, sizeof(Tanulo), n, fp);

    fclose(fp);

    return 0;
}
```

A visszaolvasáskor először beolvassuk az elemek számát. Ezután dinamikusan helyet foglalunk pontosan ennyi `Tanulo` számára, majd beolvassuk a tömböt.

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

typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;

int main() {
    FILE *fp;
    Tanulo *t;
    int n;

    fp = fopen("tanulok.dat", "rb");

    if (fp == NULL) {
```

```
        printf("Nem sikerult megnyitni az allomanyt.\n");
        return 1;
    }

    if (fread(&n, sizeof(int), 1, fp) != 1) {
        printf("Nem sikerult beolvasni az elemszamot.\n");
        fclose(fp);
        return 1;
    }

    t = malloc(n * sizeof(Tanulo));

    if (t == NULL) {
        printf("Sikertelen memoriafoglalas.\n");
        fclose(fp);
        return 1;
    }

    if (fread(t, sizeof(Tanulo), n, fp) != n) {
        printf("Nem sikerult beolvasni a tombot.\n");
        free(t);
        fclose(fp);
        return 1;
    }

    for (int i = 0; i < n; i++) {
        printf("%s %i\n", t[i].nev, t[i].pontszam);
    }

    free(t);
    fclose(fp);

    return 0;
}
```

Ez a példa összekapcsol több korábbi témát: struktúrákat, tömböket, mutatókat, dinamikus memóriafoglalást és állománykezelést.

Figyeljük meg, hogy a tömb kiírásakor nem használunk ciklust:

```
fwrite(t, sizeof(Tanulo), n, fp);
```

Ez azért működik, mert a tömb elemei egymás után helyezkednek el a memóriában. A `t` kifejezés az első elem címét adja, az `fwrite` pedig `n` darab `Tanulo` méretű elemet ír ki.

15.8. Közvetlen hozzáférés bináris állományban

Szöveges állományoknál általában elejétől végéig haladunk. Bináris állományoknál azonban gyakran előnyös, hogy az adatok fix méretű rekordokként szerepelnek. Ha minden rekord azonos méretű, akkor könnyen kiszámítható, hol kezdődik a k-adik rekord.

Ehhez az `fseek` függvényt használhatjuk.

```
fseek(fp, pozicio, honnan);
```

A harmadik paraméter azt mondja meg, honnan számítjuk az elmozdulást.

```
SEEK_SET    az állomány elejétől
SEEK_CUR    az aktuális pozíciótól
SEEK_END    az állomány végétől
```

Tegyük fel, hogy egy bináris állomány egymás után `Tanulo` rekordokat tartalmaz. A harmadik tanuló beolvasásához az állomány elejétől kétrekordnyit kell előrelépnünk, mert az indexelés itt is nullától indul.

```
fseek(fp, index * sizeof(Tanulo), SEEK_SET);

if (fread(&t, sizeof(Tanulo), 1, fp) == 1) {
    printf("%s %i\n", t.nev, t.pontszam);
}
```

[i] Megjegyzés. Ha az állomány elején az elemek számát is eltároltuk, akkor a rekordok nem rögtön az állomány elején kezdődnek, hanem az `int` méretű elemszám után. Ilyenkor a k-adik rekord kezdőpozíciója:

```
sizeof(int) + k * sizeof(Tanulo)
```

Ezért a megfelelő `fseek` hívás:

```
fseek(fp, sizeof(int) + index * sizeof(Tanulo), SEEK_SET);
```

A közvetlen hozzáférés nagy előnye, hogy nem kell az állomány elejétől végigolvasni minden rekordot, ha csak egy adott helyen lévő adatra van szükségünk.



AI-csepp

Kérdezd meg az AI-t!

• Egy `int` változó az `123456789` egész számot tárolja. Mekkora lesz az állomány mérete, ha szöveges, illetve ha bináris állományba írjuk ki?

• Egy `int` változó az `5` egész számot tárolja. Mekkora lesz az állomány mérete, ha szöveges, illetve ha bináris állományba írjuk ki?
• Más lesz-e az állomány tartalma, ha a `'7'` karaktert szöveges, illetve ha bináris állományba írjuk ki?
• Más lesz-e az állomány tartalma, ha a `'\n'` karaktert szöveges, illetve ha bináris állományba írjuk ki?

15.9. AlgoRythmics szál: gyorsrendezés bináris állományból beolvasott rekordokon



Az előző fejezetben megismertük a C nyelv `qsort` könyvtári függvényét. A név a quicksort, vagyis gyorsrendezés szóból ered. A gyakorlatban a `qsort` mögött gyakran a gyorsrendezés valamelyik változata áll, bár a C szabvány nem írja elő, hogy a könyvtári függvénynek pontosan ezt az algoritmust kell használnia. A szabvány csak azt határozza meg, hogyan kell meghívni a függvényt, nem azt, hogy belül milyen rendezési stratégiával dolgozzon.

A gyorsrendezés az AlgoRythmics-videóban is látványos formában jelenik meg (15.1. ábra). A táncosok nem egyszerűen egymás mellett álló elemeket cserélgetnek, hanem a sort egy kiválasztott elemhez viszonyítva két részre bontják. Az egyik oldalra kerülnek a nála kisebb, a másik oldalra a nála nagyobb elemek. Ezután ugyanezt a gondolatot alkalmazzák a két részre külön-külön.



15.1. ábra. *Pillanatkép a gyorsrendezést Küküllő menti legényessel ábrázoló AlgoRythmics videóból*

▦ **Algoritmikus mintázat.** A gyorsrendezés alapgondolata az oszd meg és uralkodj elv:

```
1. Válasszunk ki egy elemet, ez lesz a vezérelem.
2. Rendezzük át a tömböt úgy, hogy a vezérelemnél kisebb elemek
   az egyik, a nagyobbak a másik oldalra kerüljenek.
3. Rendezzük gyorsrendezéssel a bal oldali részt.
4. Rendezzük gyorsrendezéssel a jobb oldali részt.
```

A gyorsrendezés tehát rekurzív algoritmus. Egy nagy rendezési feladatot két kisebb rendezési feladatra bont. Ha egy részben már nulla vagy egy elem van, akkor nincs mit rendezni.

Most kapcsoljuk ezt össze a fejezet fő témájával, az állománykezeléssel. Tegyük fel, hogy tanulók adatait bináris állományban tároltuk. A program először beolvassa a struktúratömböt, majd pontszám szerint csökkenő sorrendbe rendezi.

```
typedef struct {
    char nev[50];
    int pontszam;
} Tanulo;
```

A rendezéshez most nem a `qsort` könyvtári függvényt használjuk, hanem saját gyorsrendezést írunk. Így jobban láthatóvá válik az a stratégia, amelyről a `qsort` a nevét kapta.

Először megadjuk, mit jelent két tanuló sorrendje. A nagyobb pontszámú tanuló kerüljön előbbre.

```
int elobb_kerul_e(Tanulo a, Tanulo b) {
    return a.pontszam > b.pontszam;
}
```

Szükségünk lesz egy cserefüggvényre is. A csere most teljes `Tanulo` struktúrákat mozgat, vagyis a név és a pontszám együtt marad.

```
void tanulo_csere(Tanulo *a, Tanulo *b) {
    Tanulo seged = *a;
    *a = *b;
    *b = seged;
}
```

A gyorsrendezés egyik legfontosabb lépése a szétválasztás. Mivel az AlgoRythmics videóban a táncosok mindig a vizsgált szakasz első elemét választják vezérelemnek, a mi megoldásunkban is ezt a változatot követjük. A `szetvalaszt` függvény tehát a bal indexű elemből indul ki, és ehhez viszonyítva rendezi át a tömb adott szakaszát.

```
int szetvalaszt(Tanulo t[], int bal, int jobb) {
    Tanulo vezer = t[bal];
    int i = bal + 1;
    int j = jobb;
    while (i <= j) {
```

```
        while (i <= jobb && elobb_kerul_e(t[i], vezer)) {
            i++;
        }

        while (j > bal && elobb_kerul_e(vezer, t[j])) {
            j--;
        }

        if (i < j) {
            tanulo_csere(&t[i], &t[j]);
            i++;
            j--;
        }
    }

    tanulo_csere(&t[bal], &t[j]);

    return j;
}
```

[i] Megjegyzés. A `szetvalaszt` függvény megértése jó alkalom a kódolvasás gyakorlására. Ha első olvasásra nem világos a működése, kérhetünk AI-segítséget is: például megkérhetjük, hogy kövesse végig a függvényt egy rövid, 5-6 elemű tömbön, és mutassa meg, hogyan változnak az indexek, és mikor történnek cserék.

A `szetvalaszt` függvény visszatérési értéke annak az elemnek az indexe, amely a szétválasztás után már a végleges helyén áll. Ettől balra olyan elemek vannak, amelyeknek előbb kell sorra kerülniük, ettől jobbra pedig olyanok, amelyeknek később.

Erre épül maga a rekurzív gyorsrendezés:

```
void gyorsrendez(Tanulo t[], int bal, int jobb) {
    if (bal < jobb) {
        int hely = szetvalaszt(t, bal, jobb);

        gyorsrendez(t, bal, hely - 1);
        gyorsrendez(t, hely + 1, jobb);
    }
}
```

Az alábbi kivonat csak a lényegi lépéseket mutatja. A hibakezelést, az állományok lezárását és a lefoglalt memória felszabadítását most elhagytuk, hogy jobban látszódjon az adatfeldolgozás fő menete: beolvasás bináris állományból,

rendezés gyorsrendezéssel, majd a rendezett adatok visszaírása egy másik bináris állományba.

```
FILE *fp1 = fopen("tanulok.dat", "rb");

int n;
fread(&n, sizeof(int), 1, fp1);

Tanulo *tanulok = malloc(n * sizeof(Tanulo));
fread(tanulok, sizeof(Tanulo), n, fp1);

gyorsrendez(tanulok, 0, n - 1);

FILE *fp2 = fopen("rendezett_tanulok.dat", "wb");

fwrite(&n, sizeof(int), 1, fp2);
fwrite(tanulok, sizeof(Tanulo), n, fp2);
```

Ebben a megoldásban ismét szétválasztottunk három gondolatot: (1) mit jelent két elem sorrendje? (2) hogyan cserélünk ki két elemet? (3) milyen rendezési mintát használunk? Ez ugyanaz a programozói gondolat, amelyet a `qsort` esetében is láttunk. A rendezés általános mintája különválik attól, hogy pontosan milyen típusú elemeket rendezünk, és milyen szempont szerint döntjük el a sorrendjüket.

Időbonyolultság. A gyorsrendezés átlagos esetben nagyon hatékony, időbonyolultsága $O(n \log n)$. Rosszul megválasztott vezérelemek esetén azonban előfordulhat, hogy a szétválasztás nagyon egyenetlen lesz, és az algoritmus $O(n^2)$ időben fut. Ezért a valódi könyvtári rendezőfüggvények gyakran finomított változatokat használnak, például ügyesebb vezérelemválasztást vagy több rendezési stratégia kombinációját.



AI-csepp

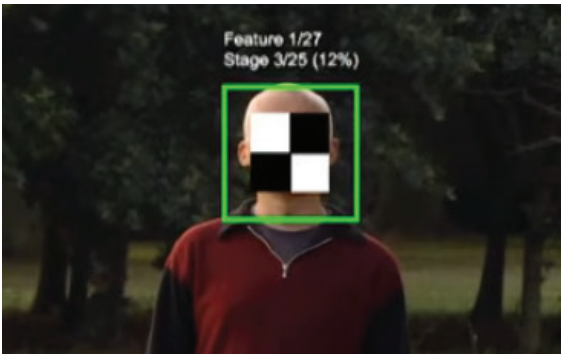
Kérdezd meg az AI-t!

- Hogyan lehetne módosítani az `elobb_kerul_e` függvényt, ha nem pontszám szerint csökkenő, hanem név szerint ábécérendbe szeretnénk rendezni a tanulókat?
- Mi történik, ha a vezérelem mindig nagyon rosszul választódik meg?
- Miért lehet a gyorsrendezés átlagosan gyors, miközben legrosszabb esetben $O(n^2)$ időbonyolultságú?
- Miben hasonlít és miben különbözik a `gyorsrendez` függvényünk a 12.4.3. alfejezetben bemutatott `osszefesuleses_rendez` függvénytől?

15.10. BBC szál: arcfelismerés, képfeldolgozás és bináris képállományok



A BBC-film az arcfelismerés példáján keresztül is megmutatja, hogy a számítógép olyan adatokat is számokként kezel, amelyek számunkra képeknek, arcoknak, alakzatoknak vagy mintázatoknak tűnnek (15.2. ábra). Amikor mi egy arcot látunk, szemeket, orrot, szájat, árnyékokat és arányokat érzékelünk. A program számára azonban a kép képpontokból áll, a képpontok pedig számokkal leírható színértékeket hordoznak.



15.2. ábra. *A BBC filmje az arcfelismerés példáján keresztül vezeti be a nézőt a számítógépes algoritmusok világába*

Az arcfelismerés természetesen összetett feladat. Nem elég egyetlen képpontot megvizsgálni; mintázatokat, körvonalakat, távolságokat, arányokat és sok más jellemzőt kell figyelembe venni. Mielőtt azonban bármilyen felismerési algoritmus dolgozni kezdene, a képet be kell olvasni és a program számára feldolgozható adatszerkezetben kell tárolni.

Ez jól kapcsolódik a bináris állományokhoz. Egy képállomány nem más, mint bájtok sorozata, amely valamilyen szabály szerint írja le a képet. A valódi képformátumok, például a JPEG vagy a PNG, bonyolult szerkezetűek, gyakran tömörítést is használnak. Ezért most egy egyszerűbb formátummal, a PPM-formátum egyik változatával dolgozunk.

A PPM-formátum `P6` változata színes képet tárol. Az állomány elején egy rövid szöveges fejléc található, amely megadja a formátumot, a kép méretét és a maximális színértéket. Ezután következnek a képpontok bináris adatai.

Egy színes képpont három összetevőből áll:

```
vörös komponens
zöld komponens
kék komponens
```

Ezt gyakran RGB-ábrázolásnak nevezzük. Ha minden komponens egy `unsigned char` érték, akkor egy képpont 3 bájtot foglal. A komponensek értéke általában `0` és `255` közé esik: a `0` az adott színösszetevő hiányát, a `255` pedig a maximális intenzitását jelenti.

Ha a kép szélessége `x`, magassága pedig `y`, akkor a képpontok száma: `x * y`. Mivel minden képpont 3 bájtból áll, a képadatok mérete: `x * y * 3` bájt.

A következő kódrészlet egy `kep.ppm` nevű PPM-képet nyit meg bináris olvasásra, beolvassa a fejléc néhány sorát, majd a teljes képadatot egy dinamikusan foglalt tömbbe tölti.

```
FILE *fin;
char formatum[3];
int x, y, maxszin;
unsigned char *kep;

fin = fopen("kep.ppm", "rb");

if (fin == NULL) {
    printf("Nem sikerult megnyitni a kepet.\n");
    return 1;
}

/* magic number: P6 */
fscanf(fin, "%2s", formatum);

/* megjegyzes kihagyasa */
fscanf(fin, " %*[^\n]");
fscanf(fin, "%*c");

/* szelesseg, magassag, maximalis szinertek */
fscanf(fin, "%i %i %i", &x, &y, &maxszin);
fscanf(fin, "%*c");

kep = malloc(x * y * 3 * sizeof(unsigned char));

if (kep == NULL) {
    printf("Sikertelen memoriafoglalas.\n");
    fclose(fin);
    return 1;
```

```
}

fread(kep, sizeof(unsigned char), x * y * 3, fin);

fclose(fin);
```

[i] Megjegyzés. Bár a `kep.ppm` állományt bináris olvasásra nyitottuk meg, az elején mégis használunk `fscanf` hívásokat. Az `rb` megnyitási mód tehát nem azt jelenti, hogy az állományból csak `fread`-del lehet olvasni. Azt jelenti, hogy a program bináris módban kezeli a streamet, vagyis nem végez szöveges sorvég-átalakításokat. A szöveges fejléc beolvasására használhatunk `fscanf`-et, a képpontok bájtjainak beolvasására pedig `fread`-et.

A `kep` tömbben sorfolytonosan tároljuk a képpontokat. Ez azt jelenti, hogy először az első sor képpontjai következnek balról jobbra, majd a második sor képpontjai, és így tovább. Minden képponthoz három egymás utáni bájt tartozik: vörös, zöld és kék komponens.

A teljes képadat beolvasása egyetlen `fread` hívással történik. Ez a hívás `x * y * 3` darab `unsigned char` értéket olvas be az állományból a `kep` tömbbe.

Ha például az `i`-edik sor `j`-edik oszlopában lévő képpontot szeretnénk elérni, akkor a hozzá tartozó első bájt indexe:

```
int index = (i * x + j) * 3;
```

Ekkor a három színkomponens:

```
unsigned char r = kep[index];
unsigned char g = kep[index + 1];
unsigned char b = kep[index + 2];
```

Innen kezdve a kép már nem titokzatos állomány, hanem egy egyszerű bájttömb. A program különböző műveleteket végezhet rajta: kiszámíthatja a kép átlagos fényességét, megkeresheti a nagyon világos vagy nagyon sötét részeket, módosíthatja a színeket, vagy új állományba írhatja a megváltoztatott képet.

Az arcfelismerés is ilyen alapokon nyugszik, csak sokkal összetettebb feldolgozást végez. A program először képpontokként olvassa be a képet, majd ezekből próbál magasabb szintű jellemzőket felismerni: éleket, alakzatokat, arányokat, mintázatokat. A lényeg azonban ugyanaz: a számítógép számára a kép is adat, az adat pedig bájtok formájában olvasható, tárolható és feldolgozható.

[i] Megjegyzés. A fenti példa leegyszerűsített PPM-beolvasó. Feltételezi, hogy a fejléc pontosan három beolvasási lépéssel feldolgozható, és hogy a második sor megjegyzés. Valódi PPM-állományoknál a megjegyzések kezelése ennél

általánosabb lehet, ezért teljes képbeolvasó programban több ellenőrzésre lenne szükség.



AI-csepp

Kérdezd meg az AI-t!

- Mit jelent az RGB-színábrázolás, és miért tartozik egy színes képponthoz három számérték?
- Hogyan lehet kiszámítani egy adott sorban és oszlopban lévő képpont helyét az egydimenziós `kep` tömbben? Fűzz magyarázatot az indexet kiszámító képlethez!
- Hogyan lehetne a beolvasott képet szürkeárnyalatossá alakítani?
- Miben különbözik egy egyszerű PPM-kép beolvasása egy tömörített JPEG- vagy PNG-kép beolvasásától?

15.11. Hiúz szál: egyensúlyi pontok a vadkamera felvételeiben



A hiúzrezervátum egyik erdei ösvényén vadkamera működik. A kamera minden órában rögzíti, hány hiúzmozgást észlelt az adott időszakban. Az adatokat a rendszer egy `szamok.dat` nevű bináris állományba menti. Az állomány egymás után tárolt `int` típusú egész számokat tartalmaz: minden szám egy-egy időszak megfigyelési értéke.

A kutatókat most azok az időpontok érdeklik, amelyek egyensúlyi pontnak tekinthetők. Egy pozíció akkor egyensúlyi pozíció, ha az előtte lévő időszakokban összesen ugyanannyi hiúzmozgást figyeltek meg, mint az utána következő időszakokban.

Az aktuális pozíción álló érték egyik összegbe sem számít bele. A pozíciók számozása 0-tól kezdődik.

A számsorról annyit tudunk, hogy elemeinek összege belefér a `long long` típus értéktartományába. A program írja ki az összes egyensúlyi pozíciót növekvő sorrendben, soronként egyet. Ha nincs ilyen pozíció, akkor írja ki: `NINCS`.

Például ha az állományban a `3 1 2 5 2 1 0 2 1` számok szerepelnek, akkor a 3 pozíció egyensúlyi pozíció, mert az előtte lévő értékek összege és az utána lévő értékek összege is 6.

[i] Megjegyzés. A feladatban szerepel, hogy az értékek összege belefér a `long long` típus értéktartományába. Ez arra is utalhat, hogy az állomány akár olyan nagy lehet, hogy a teljes számsorozat memóriában való eltárolása nem életszerű. Ezért olyan megoldásban gondolkodunk, amely egyszerre mindig csak az aktuális elemet olvassa ki az állományból.

▦ **Algoritmikus mintázat.** Első ötletként minden pozíciónál külön kiszámíthatnánk a bal oldali és a jobb oldali összeget. Ez azonban sok ismételt olvasással járna. Hatékonyabb gondolat, ha először meghatározzuk az összes szám összegét, majd balról jobbra haladva fenntartjuk a bal oldali összeget.

Ha az aktuális szám értéke `x`, akkor az aktuális pozíciónál:

```
bal oldali összeg  = bal
jobb oldali összeg = teljes_osszeg - bal - x
```

A pozíció akkor egyensúlyi pozíció, ha ez a két érték megegyezik.

Az alábbi kódrészlet szerint az állományt kétszer járjuk be. Az első bejárás kiszámítja a teljes összeget, a második pedig megkeresi az egyensúlyi pozíciókat. Így nincs szükség arra, hogy az egész számsort tömbbe olvassuk.

```
FILE *fp = fopen("szamok.dat", "rb");

int x;
long long teljes_osszeg = 0;
long long bal;
long long jobb;
long long pozicio;
int talalt;

while (fread(&x, sizeof(int), 1, fp) == 1) {
    teljes_osszeg += x;
}

fseek(fp, 0, SEEK_SET);

bal = 0;
pozicio = 0;
talalt = 0;

while (fread(&x, sizeof(int), 1, fp) == 1) {
    jobb = teljes_osszeg - bal - x;

    if (bal == jobb) {
        printf("%lli\n", pozicio);
        talalt = 1;
    }

    bal += x;
    pozicio++;
```

```
}

if (!talalt) {
    printf("NINCS\n");
}

fclose(fp);
```

Ez a megoldás jól mutatja, hogy bináris állományokkal nemcsak úgy dolgozhatunk, hogy mindent egyszerre beolvasunk a memóriába. Az `fseek` segítségével az állományban mozoghatunk, majd több bejárással és kevés memória használatával is megoldhatunk egy logikai feladatot.

[i] Megjegyzés. Az itt javasolt megoldás időben lineáris: az állományt kétszer járja be, ezért az időbonyolultsága `O(n)`. Ez algoritmikus szempontból hatékony, hiszen nem számítjuk újra minden pozíciónál külön a bal és jobb oldali összeget. A lineáris idő azonban nem jelenti azt, hogy tetszőlegesen nagy állományok feldolgozása gyors lenne. Ha az állomány több milliárd vagy még több rekordot tartalmaz, akkor már maga az állomány végigolvasása is jelentős időt vehet igénybe. Ilyenkor nem az algoritmus rendje „száll el”, hanem az adatmennyiség válik fizikailag hatalmassá.



AI-csepp
Kérdezd meg az AI-t!
• Még ha az `O(n)` időbonyolultság nem is javítható, lehet-e gyorsítani a programon a gyakorlatban?

15.12. ECN szál: amikor a beolvasás és kiírás a feladat része



Programozási versenyeken a bemenet és a kimenet szinte mindig állományszerűen jelenik meg: a program sorban olvassa az adatokat, majd pontosan előírt formában írja ki az eredményt. Ez elsőre technikai részletnek tűnhet, de sokszor már az adatok helyes beolvasása is komoly figyelmet igényel.

Jó példa erre az egyik 2019-es ECN-feladat, az *Open data about train timetable*. A feladat története szerint Landia vasúttársasága nyílt adatként közzéteszi a menetrendi információkat, de az adatokban többféle ellentmondás is előfordul. A programnak azokat az útvonalakat kell megtalálnia, ahol eltérés van az oda- és visszairányú vonatok száma, illetve a közlekedési korlátozások között.

Ebben a fejezetben nem oldjuk meg a teljes feladatot, most csak az adatok beolvasására összpontosítunk. Ha az olvasót érdekli a versenyfeladat teljes megoldása, érdemes önállóan továbbgondolnia, hogyan lehet az útvonalakat párokba rendezni, összehasonlítani, majd a kért sorrendben kiírni.

A bemenet első sora az állomások és a vonatok számát tartalmazza. Ezután következnek az állomások adatai: minden állomásnak van egy egész kódja és egy neve. Az állomásnév szóközt is tartalmazhat, ezért beolvasása külön figyelmet igényel.

Ezután következnek a vonatok adatai, soronként hat mezővel: vonatkód, korlátozás, induló állomás kódja, indulási idő, érkező állomás kódja, érkezési idő. A csavar az, hogy ezek közül nem minden mezőre van szükség. A vonatkódot, az indulási időt és az érkezési időt be kell olvasni, mert szerepelnek a bemenetben, de a megoldás lényegi részében nem kell felhasználni őket.

Példabemenet:

```
4 9
15 Paris Est
11 Roma Ovest
12 Bucuresti Nord
13 Budapest Keleti
101 0 15 7:00 11 18:00
102 0 11 8:00 15 20:00
103 1 15 9:00 11 21:00
201 0 12 6:00 13 23:00
202 1 13 5:00 12 22:00
301 0 11 6:00 13 23:30
302 0 13 5:00 11 22:30
203 0 12 12:00 13 10:00
204 0 13 15:00 12 12:00
```

Példakimenet:

```
12 Bucuresti Nord - 13 Budapest Keleti, restrictions
15 Paris Est - 11 Roma Ovest, number of trains
```

Az állomások és vonatok adatainak eltárolásához célszerű lehet az alábbi két `struct` típust definiálni:

```
typedef struct {
    int kod;
    char nev[28];
} Allomas;
```

```
typedef struct {
    int honnan;
    int hova;
    int korlatozas;
} Vonat;
```

Ezek után a bemeneti adatok beolvasása az alábbi módon történhet:

```
int allomasok_szama;
int vonatok_szama;

fscanf(fin, "%i %i", &allomasok_szama, &vonatok_szama);

Allomas allomasok[40000];

for (int i = 0; i < allomasok_szama; i++) {
    fscanf(fin, "%i ", &allomasok[i].kod);
    fscanf(fin, "%27[^\n]", allomasok[i].nev);
    fscanf(fin, "%*c");
}

Vonat vonatok[160000];

for (int i = 0; i < vonatok_szama; i++) {
    fscanf(fin, "%*s %i %i %*s %i %*s",
           &vonatok[i].korlatozas,
           &vonatok[i].honnan,
           &vonatok[i].hova);
}
```

Az állomások beolvasásakor a `%27[^\n]` legfeljebb 27 karaktert olvas be a sor végéig. Ez illeszkedik ahhoz, hogy a feladat szerint az állomásnév hossza legfeljebb 27 karakter. A következő `fscanf` a sorvége karaktert dobja el.

A vonatok beolvasásakor használtuk a `*` jelet. Emlékezzünk, hogy ez azt jelenti: az adott mezőt a program beolvassa, de nem tárolja el változóba. Így a vonatkódot, az indulási időt és az érkezési időt átugorjuk, miközben a szükséges három adat – a korlátozás, az induló állomás kódja és az érkező állomás kódja – bekerül a `Vonat` struktúrába.

A kimenet formázása hasonlóan fontos. Sok feladatnál például az eredményeket szóközzel elválasztva kell kiírni, de az utolsó érték után már nem szerepelhet fölösleges szóköz. Ha nem tudjuk előre, melyik lesz az utolsó kiírt elem, akkor ennek külön kezelése nem mindig magától értetődő.

Ilyenkor jobb minta, ha nem az érték után írjuk ki az elválasztót, hanem az érték elé – de csak a második kiírt elemtől kezdve. Az első kiírt értéket általában könnyebb felismerni és rendhagyóan kezelni, mint az utolsót.

```
int elso = 1;

for (int i = 0; i < n; i++) {
    if (ki_kell_irni(i)) {
        if (!elso) {
            printf(" ");
        }

        printf("%i", ertek[i]);
        elso = 0;
    }
}

printf("\n");
```

Ez a kis minta sok versenyfeladatban hasznos. Az `elso` változó azt jelzi, hogy írtunk-e már ki korábban értéket. Az első érték elé nem kerül szóköz, minden további érték elé viszont igen. Így az értékek között pontosan egy szóköz lesz, az utolsó után pedig nem marad fölösleges elválasztó.



AI-csepp

Kérdezd meg az AI-t!

- Milyen tipikus hibák fordulhatnak elő programozási versenyeken állományból való beolvasáskor?
- Milyen tipikus hibák fordulhatnak elő programozási versenyeken az eredmények kiírásakor?
- Hogyan érdemes tesztelni, hogy a program pontosan a feladat által elvárt formátumban írja ki az eredményt?

►► Zárógondolat

Ebben a fejezetben a programjaink kiléptek a futás idejének korlátai közül. Az állománykezelés segítségével az adatok nemcsak a program memóriájában létezhetnek, hanem elmenthetők, később visszaolvashatók és újra feldolgozhatók.

Láttuk, hogy az állományok nem csupán „adatkupacok”. Mindig tudnunk kell, milyen szerkezetben tárolják az adatokat: szövegként, bináris értékekként, rekordok sorozataként vagy akár képpontok bájtjaiként. Ugyanakkor a beolvasás és a kiírás pontossága legalább olyan fontos, mint maga az algoritmus: egy kihagyott

mező, egy rosszul kezelt sorvég vagy egy fölösleges szóköz az eredmény végén hibássá teheti a programot.

Az állománykezelés tehát nem pusztán technikai kiegészítés. Ettől kezdve a programjaink képesek adatokat megőrizni, nagyobb adathalmazokat feldolgozni, képeket vagy rekordokat kezelni és pontosan kommunikálni a külvilággal. A program már nemcsak számol, hanem olvas, ír, emlékezik és adatokat cserél.

16. Mesterfejezet: haladó eszközök, mélyebb rétegek

Az előző öt fejezetben megismerkedtünk a függvényekkel, a rekurzióval, a karakterláncokkal, a struktúrákkal és az állománykezeléssel. Ezek önmagukban is fontos témák, de a C igazi ereje akkor jelenik meg, amikor ezeket együtt kezdjük használni.

Ez a mesterfejezet egyfajta kitekintés: néhány olyan haladóbb eszközt és összefüggést mutat be, amelyek segítenek mélyebben megérteni a korábbi fejezeteket. A cél nem az, hogy minden részletet kimerítően tárgyaljunk, hanem hogy meglássuk: a C-ben sok egyszerű eszközből épülnek fel a rugalmasabb, általánosabb megoldások.

További részletek végett lásd a (Kátai 2008) jegyzet megfelelő fejezeteit.

16.1. Függvénypointerek: amikor műveletet adunk át

A függvényekről szóló fejezetben azt láttuk, hogy egy függvénynek adatokat adhatunk át: egész számot, tömböt, karakterláncot, struktúrát vagy pointert. Van azonban egy különösen érdekes eset: amikor nem adatot, hanem *műveletet* adunk át.

A **`qsort`** használatakor már találkoztunk ezzel, csak akkor még nem bontottuk ki részletesen. A `qsort` negyedik paramétere egy összehasonlító függvény:

```
qsort(t, n, sizeof(Tanulo), hasonlit);
```

Itt a `hasonlit` nem egyszerű adat, hanem egy függvény neve. A `qsort` ezt a függvényt fogja meghívni, amikor két elemet össze kell hasonlítania.

A `qsort` negyedik paraméterének típusa lényegében ez:

```
int (*)(const void *, const void *)
```

Ez egy függvénypointer-típus. Olyan függvény címét tárolhatja, amely két `const void *` paramétert kap, és `int` értékkel tér vissza.

Egy ilyen összehasonlító függvény például:

```
int pontszam_szerint(const void *a, const void *b) {
    const Tanulo *pa = a;
    const Tanulo *pb = b;
```

```
    if (pa->pontszam < pb->pontszam) {
        return -1;
    }

    if (pa->pontszam > pb->pontszam) {
        return 1;
    }

    return 0;
}
```

A rendező algoritmus ugyanaz marad. Ami változik, az az összehasonlítás szabálya. Ez a függvénypointerek egyik legfontosabb szerepe: lehetővé teszik, hogy ugyanazt az algoritmust többféle viselkedéssel használjuk.

Először nézzünk egy egyszerűbb példát egész számokra. Írjunk olyan rendezőfüggvényt, amely nem maga dönti el, milyen sorrendbe kell rendezni, hanem paraméterként kapja meg az összehasonlítás szabályát.

```
int novekvo_rossz_sorrend(int a, int b) {
    return a > b;
}

int csokkeno_rossz_sorrend(int a, int b) {
    return a < b;
}
```

Mindkét függvény azt mondja meg, hogy két egymás melletti elem rossz sorrendben van-e. Ha igen, cserélni kell őket.

```
void rendez(int t[], int n, int (*rossz_sorrend)(int, int)) {
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - 1 - i; j++) {
            if (rossz_sorrend(t[j], t[j + 1])) {
                int seged = t[j];
                t[j] = t[j + 1];
                t[j + 1] = seged;
            }
        }
    }
}
```

A harmadik paraméter:

```
int (*rossz_sorrend)(int, int)
```

azt jelenti, hogy a `rossz_sorrend` egy függvénypointer. Olyan függvényre mutathat, amely két `int` paramétert kap, és `int` értékkel tér vissza.

A rendezőfüggvény így hívható meg:

```
rendez(t, n, novekvo_rossz_sorrend);
```

vagy:

```
rendez(t, n, csokkeno_rossz_sorrend);
```

A `rendez` függvényt nem kellett újraírni. Csak azt cseréltük ki, hogy milyen szabály alapján döntsön a sorrendről.

A `qsort` azért tud bármilyen típusú tömböt rendezni, mert nem ismeri a tömb elemeinek konkrét típusát. Ehelyett `void *` pointerekkel és az elemmérettel dolgozik. Ugyanez az ötlet saját általános keresőfüggvénynél is használható.

Egy leegyszerűsített lineáris kereső alakja például ilyen lehet:

```
void *keres(void *t, int n, int meret,
            void *kulcs,
            int (*egyezik)(const void *, const void *)) {
    char *p = t;

    for (int i = 0; i < n; i++) {
        if (egyezik(p + i * meret, kulcs)) {
            return p + i * meret;
        }
    }

    return NULL;
}
```

Itt a `keres` függvény nem tudja, hogy egész számokat, karakterláncokat vagy struktúrákat keresünk. Csak azt tudja, hogy:

- hol kezdődik a tömb;
- hány elem van benne;
- hány bájtos egy elem;
- hol van a keresett kulcs;
- milyen függvénnyel kell eldönteni, hogy két elem egyezik-e.

Ez ugyanaz a gondolat, mint a `qsort` esetében: az algoritmus általános, a konkrét szabályt pedig függvényként adjuk át.

16.2. Változó paraméterszámú függvények

A `printf` és a `scanf` különlegesnek tűnhetnek: nem mindig ugyanannyi paramétert kapnak.

```
printf("%i\n", x);
printf("%i %i %i\n", a, b, c);
```

A C lehetővé teszi változó paraméterszámú függvények írását is. Ezeknél a paraméterlista végén három pont szerepel:

```
void pelda(const char *formatum, ...);
```

A három pont azt jelzi, hogy a függvény további paramétereket is kaphat, de ezek száma és típusa nem látszik közvetlenül a függvény fejlécéből.

Egy saját `myscanf` vagy `myprintf` megírása már haladó feladat, mert a változó paraméterek kezeléséhez a **`stdarg.h`** eszközeit kell használni: **`va_list`**, **`va_start`**, **`va_arg`**, **`va_end`**.

Ebben a könyvben ezt csak kitekintésként említjük meg. A lényeg az, hogy a `printf` és a `scanf` nem varázslat: a C nyelv ad olyan eszközt, amellyel változó paraméterszámú függvények írhatók.

16.3. A `main` paraméterei

Eddig a `main` függvényt többnyire így írtuk:

```
int main() {
    ...
}
```

A `main` függvénynek azonban lehetnek paraméterei is. Ezek a parancssori argumentumok elérésére szolgálnak.

```
int main(int argc, char *argv[]) {
    ...
}
```

Az **argc** azt adja meg, hány parancssori argumentumot kapott a program, az **argv** pedig karakterláncok tömbje: az argumentumok szöveges alakját tartalmazza.

Például ha a programot így indítjuk:

```
program adatok.txt eredmeny.txt
```

akkor:

```
argc == 3
argv[0] == "program"
argv[1] == "adatok.txt"
argv[2] == "eredmeny.txt"
```

Ez különösen hasznos állománykezelésnél. A program nem feltétlenül a kódba írt állománynevekkel dolgozik, hanem indításkor megkaphatja, melyik bemeneti és kimeneti állományt használja.

16.4. Indirekt rekurzió

A rekurzióról szóló fejezetben olyan függvényeket láttunk, amelyek közvetlenül önmagukat hívták meg. Ezt közvetlen rekurziónak nevezzük.

Van azonban indirekt rekurzió is. Ilyenkor nem egy függvény hívja meg közvetlenül önmagát, hanem több függvény hívási láncán keresztül jutunk vissza ugyanahhoz a függvényhez.

```
void A(int n);
void B(int n);

void A(int n) {
    if (n > 0) {
        printf("A: %i\n", n);
        B(n - 1);
    }
}

void B(int n) {
    if (n > 0) {
        printf("B: %i\n", n);
        A(n - 1);
    }
}
```

Itt az `A` függvény meghívja a `B` függvényt, a `B` pedig visszahívhatja az `A` függvényt. A rekurzió tehát nem egyetlen függvényen belül látszik, hanem a függvények kapcsolatában.

[i] Megjegyzés. Indirekt rekurziónál is ugyanúgy szükség van leállási feltételre. Ha a hívási láncban nincs olyan eset, amikor a függvények már nem hívják tovább egymást, akkor a program végtelen rekurzióba kerülhet.

16.5. Karakterláncok: sorok, mezők és memóriaterületek

A `string.h` függvényei közül korábban főként a legismertebbeket használtuk: `strlen`, `strcmp`, `strcpy`, `strcat`, `strchr`, `strstr`. A C könyvtár azonban további eszközöket is ad szövegek és memóriaterületek feldolgozására.

Az **`sscanf`** úgy működik, mint a `scanf`, de nem a billentyűzetről, hanem egy karakterláncból olvas.

```
char sor[] = "Anna 17 9.75";
char nev[30];
int eletkor;
float atlag;

sscanf(sor, "%29s %i %f", nev, &eletkor, &atlag);
```

A **`sprintf`** ennek fordítottja: formázott szöveget ír egy karaktertömbbe.

```
char sor[100];
int pont = 87;

sprintf(sor, "Pontszam: %i", pont);
```

[i] Megjegyzés. A `sprintf` használatakor figyelni kell arra, hogy a célként megadott tömb elég nagy legyen. Biztonságosabb változata a **`snprintf`**, amelynél megadhatjuk a célterület méretét is.

[i] Megjegyzés. Bár elvileg az `sscanf` és az `sprintf` is használható konverziós célokra, általános esetben nem ezek az ajánlott megoldások. Az `sscanf` helyett számkonverzióra célszerűbb a `strto…` típusú függvényeket használni, mert ezekkel pontosabban ellenőrizhető a hibás vagy részleges bemenet. Az `sprintf`

helyett pedig a sn`printf` ajánlott, mert figyelembe veszi a célpuffer méretét, így biztonságosabban használható.

- karakterlánc → egész szám: **`strtol, strtoul, strtoll, strtoull`**
- karakterlánc → lebegőpontos szám: **`strtof, strtod, strtold`**
- egész/lebegőpontos szám → karakterlánc: `snprintf`

A **`strtok`** egy karakterláncot darabokra bont elválasztó karakterek mentén. Például különböző elválasztó karakterekkel elválasztott adatok feldolgozására használható.

```
char sor[] = "alma, korte;szilva";
char *darab;

darab = strtok(sor, ",; ");

while (darab != NULL) {
    printf("%s\n", darab);
    darab = strtok(NULL, ",");
}
```

A `strtok` első hívásakor megadjuk a feldolgozandó karakterláncot. A következő hívásokban már `NULL` szerepel az első paraméter helyén, mert a függvény belsőleg megjegyzi, hol tartott.

[i] Megjegyzés. A `strtok` módosítja az eredeti karakterláncot: az elválasztó karakterek helyére lezáró `'\0'` karaktereket ír. Ezért csak olyan karaktertömbön használjuk, amelyet szabad módosítani. Karakterlánc-literálon nem szabad használni.

A **`memcpy, memmove, memcmp`** és **`memset`** nem karakterláncokkal, hanem nyers memóriaterületekkel dolgoznak. Ez fontos különbség: ezek a függvények nem keresik a lezáró `'\0'` karaktert, hanem pontosan annyi bájtot dolgoznak fel, amennyit megadunk.

```
int a[5] = {1, 2, 3, 4, 5};
int b[5];

memcpy(b, a, 5 * sizeof(int));
```

A `memcpy` memóriaterületet másol. Akkor használható biztonságosan, ha a forrás és a cél nem fedik át egymást. Ha átfedés lehetséges, akkor a `memmove` a megfelelőbb választás.

```
memmove(a + 1, a, 4 * sizeof(int));
```

A `memset` egy memóriaterület bájtjait állítja be egy adott értékre.

```
int latogatott[100];

memset(latogatott, 0, 100 * sizeof(int));
```

A `memcmp` két memóriaterület bájtjait hasonlítja össze.

[i] Megjegyzés. A `mem...` függvények nagyon erősek, de óvatosan kell használni őket. Nem típusokban gondolkodnak, hanem bájtokban.

16.6. Típusok szervezése: `typedef`, `enum`, `union`

A **`typedef`** nemcsak struktúráknál használható. Általánosabban arra szolgál, hogy egy típusnak új nevet adjunk.

```
typedef unsigned long Meret;
```

Ezután a `Meret` típust ugyanúgy használhatjuk, mint bármely más típust:

```
Meret fajlmeret;
```

A `typedef` különösen hasznos lehet pointer- vagy függvénypointer-típusoknál, mert olvashatóbbá teszi a bonyolult deklarációkat.

```
typedef int (*Hasonlito)(const void *, const void *);
```

Ezután egy összehasonlító függvénypointert így is deklarálhatunk:

```
Hasonlito h;
```

Az **`enum`** felsorolt értékekből álló típust ad. Akkor hasznos, ha egy változó néhány jól meghatározott állapot egyikét veheti fel.

```
typedef enum {
    URES,
    FOGLALT,
    TOROLT
} Allapot;
```

Ezután:

```
Allapot a = FOGLALT;
```

Az `enum` olvashatóbbá teszi a programot, mint ha mindenhol puszta számokat használnánk.

```
if (a == FOGLALT) {
    printf("A hely foglalt.\n");
}
```

Union és bitmező:

A **`union`** hasonlít a struktúrához, de míg a struktúra minden mezője külön memóriaterületet kap, a `union` mezői ugyanazon a memóriaterületen osztoznak. Ez azt jelenti, hogy egy időben csak az egyik mező értelmezhető biztonságosan.

```
typedef union {
    int egesz;
    float valos;
    char szoveg[20];
} Ertek;
```

A `union` akkor hasznos, ha egy adat többféle típusú lehet, de egyszerre csak az egyik értelmezésre van szükség.

A *bitmezők* olyan struktúramezők, amelyeknél megadhatjuk, hány biten tárolódjanak.

```
typedef struct {
    unsigned int olvasott : 1;
    unsigned int surgos : 1;
    unsigned int archivalt : 1;
} Jelzok;
```

Ez memóriatakarékos lehet, ha sok logikai jellegű állapotot szeretnénk tárolni.

[i] Megjegyzés. A bitmezők gépközeli eszközök. Használatuknál figyelni kell arra, hogy a pontos memóriabeli elrendezés függhet a fordítótól és a gépi környezettől.

16.7. Állománykezelés: karakterek, sorok, pozíciók

Az állománykezelésről szóló fejezetben főként a `fprintf`, `fscanf`, `fread` és `fwrite` függvényeket használtuk. A C könyvtár azonban többféle szintet kínál az állományok feldolgozására.

Karakterenkénti olvasásra és írásra használható:

```
int c;

c = fgetc(fp);
fputc(c, fp2);
```

Az `fgetc` visszatérési típusa int, nem `char`, mert az állomány vége esetén az `EOF` értéket kell tudnia visszaadni.

Soronkénti olvasásra és írásra használható:

```
char sor[200];

fgets(sor, 200, fp);
fputs(sor, fp2);
```

Az `fgets` legfeljebb a megadott méretnél eggyel kevesebb karaktert olvas be, és a karakterláncot lezárja `'\0'` karakterrel. Ha a sorban belefér, a sorvégi `'\n'` karaktert is megtartja.

A konzolos beolvasás és kiírás esetén is találkozunk hasonló függvényekkel.

```
int c = getchar();
putchar(c);
```

A **`puts`** egy karakterláncot ír ki, és a végére sortörést tesz.

```
puts("Hello");
```

A **`gets`** régebbi C programokban előfordulhat, de nem biztonságos, mert nem tudja ellenőrizni, mekkora a célként megadott tömb. Ezért kerülni kell. Helyette használjunk `fgets`-t.

```
char sor[100];

fgets(sor, 100, stdin);
```

Az állományon belüli pozíció lekérdezésére és módosítására használható az **ftell**, az **fseek** és a **rewind**.

```
long hely = ftell(fp);
```

Az `ftell` megadja az aktuális állománypozíciót. Az `fseek` segítségével más pozícióra ugorhatunk, a `rewind` pedig az állomány elejére állítja vissza a pozíciót.

```
rewind(fp);
```

Állományok átnevezésére és törlésére használható:

```
rename("regi.txt", "uj.txt");
remove("torlendo.txt");
```

A **fflush** a pufferelt kiírás kiürítésére szolgál.

```
fflush(fp);
```

[i] Megjegyzés. A legtöbb kimeneti állomány esetén a `fclose` automatikusan gondoskodik a puffer kiírásáról. A `fflush` akkor lehet hasznos, ha a program futása közben szeretnénk biztosítani, hogy az addigi kimenet ténylegesen kikerüljön az állományba vagy a képernyőre.

16.8. Az előfordító szerepe: `include`, `define` és makrók

A C nyelv egyik sajátossága, hogy a tényleges fordítást megelőzi egy előkészítő lépés, amelyet előfordításnak nevezünk. Ezt az előfordító, más néven prekompilátor vagy preprocesszor végzi. Az előfordító a program forráskódját még azelőtt dolgozza fel, hogy a fordító valódi C utasításként értelmezné azt. Feladata nem a program futási logikájának végrehajtása, hanem a forráskód előkészítése: fájlokat illeszt be, szövegrészeket helyettesít, illetve bizonyos feltételek alapján eldönti, hogy a kód mely részei kerüljenek ténylegesen fordításra.

Az egyik legismertebb előfordítási direktíva az **#include**, amellyel már korábban is találkoztunk. Ennek segítségével fejléceket, például szabványos könyvtárakhoz tartozó header fájlokat kapcsolhatunk a programhoz. Ilyen például a `#include <stdio.h>`, amely lehetővé teszi többek között a `printf`, a `scanf` vagy a `getchar` használatát. A `#include` lényegében azt eredményezi, hogy a megadott fejléc tartalma a fordítás előtt bekerül a programba, vagyis a fordító már az így kiegészített forráskódot kapja meg.

Szintén fontos direktíva a **#define**, amelyet gyakran szimbolikus konstansok létrehozására használunk. Ilyenkor egy névhez rendelünk egy értéket, például `#define MAX 100`, majd a programban a `MAX` névvel hivatkozhatunk erre az értékre. Ennek előnye, hogy a kód olvashatóbbá és könnyebben módosíthatóvá válik: ha később változtatni kell az értéken, elegendő egyetlen helyen átírni.

A `#define` azonban nemcsak egyszerű konstansok megadására alkalmas, hanem makrók létrehozására is. A makró lényegében fordítás előtti szöveghelyettesítés: amikor az előfordító találkozik a makró nevével, a helyére behelyettesíti a megadott kifejezést vagy kódrészletet. Léteznek paraméter nélküli és paraméteres, más néven függvényszerű makrók is. Egy függvényszerű makró első ránézésre valódi függvényhívásnak tűnhet, mert neve és argumentumai vannak, például `SQUARE(x)`. Valójában azonban nem futási idejű függvényhívás történik, hanem az előfordító már fordítás előtt elvégzi a helyettesítést.

Például a `#define SQUARE(x) ((x) * (x))` makró használatakor a `SQUARE(a)` kifejezés helyére fordítás előtt az `((a) * (a))` szöveg kerül. Ez egyszerű esetben jól működik, de fontos megérteni, hogy a makró nem rendelkezik a függvényekre jellemző típusellenőrzéssel, és az argumentumai akár többször is kiértékelődhetnek. Ha például egy makró paramétereként `i++` szerepel, akkor előfordulhat, hogy ez a növelés nem egyszer, hanem többször történik meg. Ez az egyik oka annak, hogy a függvényszerű makrók használata körültekintést igényel.

A makrók biztonságosabb használata érdekében a paramétereket és gyakran az egész kifejezést is zárójelezni szokás. A `#define SQUARE(x) x * x` alak például hibás eredményhez vezethet, ha az argumentum összetettebb kifejezés, például `a + b`. Ilyenkor a helyettesítés után a műveleti sorrend miatt nem feltétlenül azt kapjuk, amit várnánk. A `#define SQUARE(x) ((x) * (x))` forma ezért jobb, bár a többszöri kiértékelés problémáját ez sem szünteti meg teljesen. Ha fontos a típusbiztonság és a kiszámítható működés, sok esetben célszerűbb valódi függvényt használni.

A C nyelvben nemcsak a programozó hozhat létre makrókat, hanem léteznek előre definiált, illetve szabványos könyvtárakban található makrók is. Az előre definiált makrók közé tartozik például a `__FILE__`, amely az aktuális forrásfájl nevét adja meg, a `__LINE__`, amely az aktuális sor számát jelenti, valamint a `__DATE__` és a `__TIME__`, amelyek a fordítás dátumát és idejét tartalmazzák. Ezek különösen hasznosak lehetnek hibakeresésnél vagy naplóüzenetek készítésénél.

A szabványos könyvtárak fejléceiben függvényszerű makrók is előfordulhatnak. Ilyenek lehetnek például a **<ctype.h>** fejlécből ismert karaktervizsgáló és karakterátalakító műveletek, mint az **isdigit**, az **isalpha**, a **toupper** vagy a **tolower**. Ezeket a programozó függvényhíváshoz hasonló alakban használja, például `isdigit(c)`, de a megvalósítástól függően a háttérben lehetnek valódi függvények vagy makrók is. Ez jól mutatja, hogy C-ben a függvényszerű írásmód önmagában még nem garantálja, hogy valódi függvényt hívunk meg.

Az előfordítási direktívák közé tartoznak a feltételes fordítást vezérlő utasítások is, például az **#ifdef**, az **#ifndef**, az **#if**, az **#else** és az **#endif**. Ezek

segítségével megadható, hogy a forráskód bizonyos részei csak adott feltételek mellett kerüljenek lefordításra. Ez különösen hasznos lehet például hibakeresésnél, különböző operációs rendszerekhez írt kódrészek kezelésénél vagy header fájlok többszöri beillesztésének megakadályozásánál.

Összességében az előfordító a C fordítási folyamatának fontos előkészítő része. A `#include` segítségével külső fájlokat illeszthetünk be, a `#define` segítségével szimbolikus konstansokat és makrókat hozhatunk létre, a függvényszerű makrókkal pedig látszólag függvényszerű, valójában szöveghelyettesítésen alapuló megoldásokat írhatunk. A feltételes direktívák szabályozzák, hogy mely kódrészek kerüljenek fordításra. Ezek az eszközök hatékonyabbá és rugalmasabbá teszik a programozást, ugyanakkor különösen a makrók esetében figyelmes használatot igényelnek, mert az előfordító még a tényleges C fordítás előtt, szövegszinten avatkozik be a programba.



AI-csepp

Kérdezd meg az AI-t!

- Magyarázd el röviden, miért mondhatjuk, hogy a `qsort` nemcsak adatot, hanem viselkedést is kap paraméterként!
- Mi a különbség egy adatpointer és egy függvénypointer között?
- Hogyan segítik a függvénypointerek azt, hogy ugyanazt az algoritmust többféle szabállyal használjuk?
- Miért különlegesek a változó paraméterszámú függvények?
- Mire használhatók a `main` paraméterei, és hogyan kapcsolódnak az állománykezeléshez?
- Mi a különbség közvetlen és indirekt rekurzió között?
- Miért kell óvatosan használni a `strtok` függvényt?
- Miben különböznek a `str...` és a `mem...` kezdetű függvények?
- Mikor lehet hasznos az `enum`, a `union` vagy a bitmező?
- Mit csinál az előfordító, mielőtt a fordító ténylegesen lefordítaná a programot?
- Keress példát arra, amikor ugyanazt a feladatot meg lehet oldani makróval és függvénnyel is! Melyik megoldás biztonságosabb, és miért?

►► Zárógondolat

A mesterfejezetben nem teljesen új alapfogalmakat tanultunk, hanem a korábbi eszközöket néztük meg mélyebbről. A függvények nemcsak meghívhatók, hanem pointereken keresztül át is adhatók. A rendezés és keresés nemcsak egy konkrét típusra írható meg, hanem általánosítható. A karakterláncok nemcsak szavak tárolására valók, hanem sorok, mezők és formázott adatok feldolgozására is. A struktúrák mellett megjelentek olyan típusépítő eszközök is, mint az enum, a union és a bitmezők. Az állománykezelésnél láttuk, hogy ugyanazt az adatfolyamot karakterenként, soronként, formázva vagy binárisan is feldolgozhatjuk. Végül az

előfordító megmutatta, hogy a C program nemcsak futás közben, hanem már fordítás előtt is alakítható: fejlécek beillesztésével, makrókkal és feltételes fordítással.

Ezek az eszközök azt mutatják, hogy a C egyszerre egyszerű és mély nyelv. Kevés alapfogalomból indul ki, de ezek sokféleképpen kombinálhatók. A jó C programozó nemcsak azt tudja, melyik függvényt kell meghívni, hanem azt is érti, milyen adat, milyen cím, milyen típus és milyen memóriahasználat áll a háttérben – sőt azt is, hogy a fordítóhoz végül milyen forráskód jut el az előfordító munkája után.

17. C programozás madártávlatból: a szálak összeérnek

A könyv végéhez érve érdemes egy lépést hátralépni és felülnézetből is szemügyre venni az eddig megtett utat. Az egyes fejezetekben változókkal, feltételekkel, ciklusokkal, tömbökkel, pointerekkel, függvényekkel, struktúrákkal és állományokkal dolgoztunk, de mindezek nem elszigetelt témák voltak. A könyv felépítése tudatosan arra törekedett, hogy kevés, jól megválasztott alapfogalomból minél gazdagabb programozási szemlélet épüljön fel.

Ezt a gondolatot a Pareto-elv is kifejezi: a programozás tanulásában sem feltétlenül az a cél, hogy azonnal minden részletet birtokba vegyünk, hanem hogy először azt a viszonylag szűk eszköztárat értsük meg mélyen, amellyel a feladatok jelentős része már megoldható. A könyv ezért nem a C nyelv teljes leírására törekedett, hanem arra, hogy a legfontosabb elemekből – adatokból, vezérlésből, függvényekből, memóriakezelésből és állománykezelésből – használható, továbbépíthető alapot adjon.

A tanulási út öt szál mentén bontakozott ki. A *C szál* adta a nyelvi és technikai gerincet. A *Hiúz szál* az algoritmikus éleslátást, a feladatok mögötti logikai szerkezet felismerését erősítette. Az *AlgoRythmics szál* a keresések és rendezések világát tette láthatóvá mozgásként és ritmusként. A *BBC szál* a valós világ algoritmikus működésére nyitott ablakot, az *ECN szál* pedig versenyfeladatokon keresztül próbára tette, hogy a megszerzett tudás önálló problémamegoldássá tud-e válni.

A könyv első fejezete ezt a szemléleti keretet teremtette meg: miért érdemes ma programozni tanulni, és miért tekinthető az algoritmikus gondolkodás a 21. század egyik alapvető írástudásának. A második fejezet már közvetlenebbül az algoritmikus gondolkodás és a programozás kapcsolatát vizsgálta: hogyan lesz az emberi gondolatból lépésekre bontott eljárás, majd futtatható program.

A harmadik fejezettől kezdődött a tényleges C programozás. A változók, értékadások, kifejezések és bemeneti-kimeneti műveletek jelentették az első lépést. A negyedik fejezet előzetes térképet adott a C utasításkészletéről, megmutatva, hogy a programok vezérlése néhány alapvető szerkezettel – szekvenciával, elágazással és ismétléssel – írható le.

Az ötödiktől a nyolcadik fejezetig tartó rész az első nagy alapozó egységet alkotta. Idetartozott az elágazás, a feltételvezérelt ismétlés, a számsorozatok, az egydimenziós tömbök, a pointerek és a dinamikus memóriafoglalás. Ezek a fejezetek adták meg azt az eszköztárat, amellyel már összetettebb adatfeldolgozó programok is megírhatók.

A kilencedik és tizedik fejezet mesterfejezetként visszatekintett és elmélyített: az addig megismert adatok, kifejezések és vezérlési szerkezetek összefüggéseit vizsgálta magasabb nézőpontból. Itt már nem pusztán új utasításokról volt szó, hanem arról, hogyan állnak össze az alapfogalmak egységes programozói szemléletté.

A tizenegyediktől a tizenötödik fejezetig következett a második nagy alapozó egység: függvények, rekurzió, karakterláncok, struktúrák és állománykezelés. Ezek a témák már a programok szervezését, az adatok összefogását, a szövegek kezelését, az önhivatkozó gondolkodást és az adatok tartós megőrzését helyezték előtérbe.

A tizenhatodik fejezet ismét mesterfejezetként működött: az előző egység fogalmait kapcsolta össze, és néhány haladóbb eszközön keresztül mutatta meg, hogy a C igazi ereje az egyszerű elemek kombinálhatóságában rejlik.

Ez a zárófejezet már nem újabb nyelvi elemet vezet be. Feladata inkább az, hogy összerendezze a könyv egészét: visszatekintünk a fő fogalmakra, megmutatjuk, hogyan kapcsolódnak össze a különböző szálak, és hogyan vezet az első egyszerű programoktól az önálló, tudatos programozói gondolkodásig.

17.1. C szál: a program belső rendje

A C szál felülnézete négy kérdés köré rendeződik. Először azt vizsgáljuk meg, hogyan vezet az algoritmikus gondolkodás a program megszületéséhez; ezután azt, hogyan szerveződik a program modulokká, függvényekké és blokkokká; majd áttekintjük a típusok szerepét; végül pedig a változók világát rendezzük el élettartamuk, láthatóságuk és memóriabeli helyük szerint. Ez a négy visszatekintés nem új tananyagként, hanem rendszerezésként szolgál: segít meglátni, hogyan állnak össze a korábban tanult részletek egységes programmá.

17.1.1. Az algoritmustól a programig

Amikor számítógéppel szeretnénk megoldani egy feladatot, nem a program begépelésével kezdünk. Először meg kell értenünk magát a problémát: mit kapunk bemenetként, mit várunk eredményként, és milyen lépéseken keresztül juthatunk el az egyikből a másikba. A programozás valódi kiindulópontja ezért nem a billentyűzet, hanem az algoritmus.

Az algoritmus a megoldás terve. Megmondja, milyen műveleteket kell elvégezni, milyen sorrendben, milyen feltételek mellett, és meddig kell ismételni bizonyos lépéseket. Egy jó algoritmus olyan, mint egy jól megtervezett útvonal: nemcsak azt mondja meg, hová akarunk eljutni, hanem azt is, hogyan jutunk oda biztonságosan és hatékonyan.

A strukturált programozás abban segít, hogy ezt az útvonalat áttekinthető formában írjuk le. Alapgondolata egyszerű, mégis rendkívül erős: bonyolult feladatok megoldása is felépíthető néhány alapvető vezérlési szerkezetből. Ezek a szekvencia, az elágazás és az ismétlés. A szekvencia egymás után végrehajtott lépéseket jelent, az elágazás döntési helyzeteket kezel, az ismétlés pedig lehetővé teszi, hogy egy műveletsort többször végrehajtsunk.

Ez a három szerkezet a programozás alapvető építőanyaga. Ahogyan a mondatok szavakból, az épületek téglákból, úgy a strukturált programok is ezekből az egyszerű vezérlési formákból állnak össze. A Böhm–Jacopini-tétel éppen ennek az elméleti hátterét adja: minden olyan algoritmus, amelynek egy kezdőpontja és egy végpontja van, felépíthető szekvenciából, elágazásból és ismétlésből.

A C nyelv utasításai közvetlenül tükrözik ezt a gondolkodásmódot. Az egymás után írt utasítások szekvenciát alkotnak, az `if` és a `switch` döntéseket valósítanak meg, a `while`, a `for` és a `do while` ciklusok pedig ismétléseket írnak le. Így a strukturált programozás nem pusztán elméleti fogalom, hanem a programírás mindennapi gyakorlata.

A program tehát nem utasítások véletlenszerű halmaza, hanem tudatosan felépített szerkezet. Ha az algoritmus világos, a program is könnyebben érthető, javítható és továbbfejleszthető lesz. A strukturált programozás tehát nemcsak azt tanítja meg, hogyan írjunk működő programot, hanem azt is, hogyan írjunk rendezett, követhető és megbízható programot.

Miután láttuk, hogyan lesz a gondolatból algoritmus, majd az algoritmusból program, nézzük meg azt is, hogyan rendeződik maga a program belső egységekké.

17.1.2. A program mint szerkezet

Egy kisebb C program még könnyen áttekinthető: néhány változó, néhány utasítás, egy `main` függvény. Ahogy azonban a feladat növekszik, a program is egyre összetettebbé válik. Ilyenkor már nem elég, hogy a kód működjön; az is fontos, hogy érthető, javítható és bővíthető legyen. A moduláris programozás éppen ebben segít: rendet visz a növekvő programba.

A moduláris gondolkodás alapja a felbontás. A nagy feladatot kisebb, jól körülhatárolható részfeladatokra osztjuk, majd ezekhez külön függvényeket, nagyobb programok esetén pedig külön modulokat rendelünk. Egy modul összetartozó függvények és adatok egysége, rendszerint egy külön forrásállomány. A főmodul tartalmazza a `main` függvényt, a többi modul pedig egy-egy részterület megvalósításáért felel. A végén ezek a külön elkészített és külön lefordítható darabok az összeszerkesztés során egyetlen programmá állnak össze.

Ez a felépítés hasonlít ahhoz, ahogyan egy jól szervezett közösség vagy szervezet működik. Nem mindenki csinál mindent, hanem mindenkinek megvan a maga feladata. Egy függvény egy konkrét műveletért felel, egy modul pedig összetartozó műveletek csoportját fogja össze. Ettől a program nem szétesik, hanem

éppen ellenkezőleg: átláthatóbbá válik. Ha később hibát keresünk, bővítünk vagy tesztelünk, nem az egész programban kell egyszerre gondolkodnunk, hanem elegendő a megfelelő egységre összpontosítani.

A program belső rendjét a blokkok is erősítik. Blokknak nevezzük a kapcsos zárójelek közé zárt programrészt. Ilyen blokk egy függvény törzse, de a függvényen belül is létrehozhatunk további alblokkokat. Ezek egymásba ágyazódhatnak, és hierarchiát alkotnak. A blokk nemcsak formai egység: meghatározza azt is, hogy a benne létrehozott nevek meddig és honnan láthatók. Így a program szerkezete és a változók érvényességi köre szorosan összekapcsolódik.

A moduláris programozás másik fontos gondolata a használat és a megvalósítás szétválasztása. Ehhez kell értenünk a deklaráció és a definíció különbségét. Deklarációkor bejelentjük a fordítónak, hogy létezik egy adott nevű függvény vagy változó, és megadjuk, hogyan lehet hivatkozni rá. Definíciókor viszont ténylegesen létrehozzuk: változó esetén tárhelyet foglalunk, függvény esetén megírjuk az utasításait. Más szóval: a deklaráció ígéret, a definíció megvalósítás.

Ebben kapnak fontos szerepet a headerállományok. A fejállományok olyanok, mint egy modul névjegyei: megmutatják, milyen függvényeket kínál a modul a program többi része számára, anélkül, hogy feltárnák a teljes belső megvalósítást. A `#include` direktíva segítségével ezeket a deklarációkat logikailag beillesztjük a forráskódba, így a fordító tudni fogja, milyen függvényekre hivatkozhatunk. A megvalósítás továbbra is a megfelelő .c állományban marad.

E felülnézet legfontosabb üzenete, hogy a jó program nem egy hosszú, egybefolyó kódszöveg. Inkább egymásra épülő egységek rendszere: a program modulokból áll, a modulok függvényekből, a függvények blokkokból, a blokkok pedig definíciókból és utasításokból. Minél tudatosabban alakítjuk ki ezt a szerkezetet, annál inkább olyan programot kapunk, amely nemcsak ma működik, hanem holnap is érthető, javítható és továbbépíthető lesz.

A program szerkezete azonban nemcsak modulokból és függvényekből áll. Legalább ilyen fontos kérdés, hogy milyen adatokkal dolgozunk, és hogyan értelmezi ezeket a C nyelv.

17.1.3. A típusok összképe

A C-ben minden adatnak típusa van. A típus megmondja, hogyan kell értelmezni a memóriában tárolt biteket, milyen műveletek végezhetők az adott adattal, és mekkora helyre lehet szükség a tárolásához.

Egy `int` értéket egész számként értelmezünk, egy `float` értéket valós szám közelítéseként, egy `char` értéket karakterként vagy kis egész számként. Ugyanaz a memóriaterület más típussal más jelentést kaphat. Ezért a típus nem puszta dísz a változó neve előtt, hanem a program egyik alapvető szervezőelve.

A könyv során a típusok több rétegével találkoztunk:

```
alaptípusok          int, char, float, double
módosított típusok   unsigned int, long int, short int
tömbtípusok          int[100], char[50]
pointertípusok       int *, char *, FILE *
struktúratípusok     struct Tanulo, struct Pont
felsorolt típusok    enum Allapot
uniók                union Ertek
függvénytípusok      int (int, int)
```

A típusok segítségével a C egyszerre tud gépközeli és szervezett lenni. Közel enged a memóriához, de közben megköveteli, hogy a programozó megmondja, az adott memóriarészletet milyen értékként szeretné értelmezni.

A típusok megmutatják, hogyan értelmezzük az adatokat, de a változók viselkedéséhez még egy további nézőpont is szükséges: hol jönnek létre, meddig élnek, és honnan érhetők el.

17.1.4. Globális, lokális, statikus és dinamikus változók

A változókról eddig azt mondtuk, hogy névvel ellátott tárolóhelyek. Most ezt a képet tovább pontosítjuk. Egy változó viselkedését C-ben nemcsak a neve és a típusa határozza meg, hanem az is, hogy *hol deklaráljuk*, *meddig létezik*, és *a program mely részei férhetnek hozzá*. Ezek alapján beszélhetünk globális, lokális, statikus és dinamikus változókról.

Globális változónak nevezzük azt a változót, amelyet függvényeken kívül deklarálunk. A globális változó a program indulásakor jön létre, és a program befejezéséig létezik. Ez azt jelenti, hogy élettartama az egész program futására kiterjed.

A globális változó láthatósági tartománya alapértelmezés szerint attól a ponttól kezdődik, ahol deklaráltuk, és a forrásfájl végéig tart. Ezért ugyanabban a fájlban az utána következő függvények hozzáférhetnek.

A globális változók előnye, hogy több függvény között is közös adatot biztosítanak. Hátrányuk viszont az, hogy nehezebb követni, melyik függvény mikor módosította az értéküket. Ezért globális változókat csak indokolt esetben érdemes használni.

Jellemzőik:

```
deklaráció helye:    függvényeken kívül
élettartam:          a program teljes futása alatt
láthatóság:          a deklarációtól a fájl végéig
memóriaterület:      statikus memóriaterület
kezdőérték:          ha nem adunk meg értéket, automatikusan 0
```

Lokális változónak nevezzük azt a változót, amelyet egy függvényen vagy blokkon belül deklarálunk. A lokális változó neve csak abban a blokkban látható, ahol deklaráltuk. A blokkon kívül nem lehet közvetlenül hivatkozni rá.

A lokális változó általában akkor jön létre, amikor a program végrehajtása belép abba a blokkba, amelyben deklaráltuk, és akkor szűnik meg, amikor a blokk végrehajtása befejeződik. Ezért egy egyszerű lokális változó minden függvényhíváskor újra létrejön.

A lokális változók jellemzői:

```
deklaráció helye:   függvényen vagy blokkon belül
élettartam:         általában a blokk végrehajtásának ideje
láthatóság:         csak a saját blokkjában
memóriaterület:     általában verem, stack
kezdőérték:         ha nem adunk meg értéket, nem meghatározott
```

Ez fontos különbség a globális változókhoz képest: a globális változók automatikusan 0 kezdőértéket kapnak, az egyszerű lokális változók viszont nem.

A `static` kulcsszóval deklarált változók statikus élettartamúak. Ez azt jelenti, hogy a program indulásától a program befejezéséig léteznek.

A statikus változók két gyakori formája:

```
globális statikus változó
lokális statikus változó
```

Globális statikus változó: ha egy függvényeken kívül deklarált változó elé odaírjuk a `static` kulcsszót, akkor a változó csak az adott forrásfájlon belül lesz használható.

```
static int x = 10;
```

Ez az `x` változó továbbra is a program teljes futása alatt létezik, de más forrásfájlokból nem érhető el közvetlenül. Vagyis itt a `static` főleg a láthatóságot korlátozza.

A *lokális statikus változó* olyan változó, amelyet függvényen vagy blokkon belül deklarálunk, de `static` kulcsszóval.

```
void szamol(void) {
    static int db = 0;
    db++;
    printf("%d ", db);
}
```

A `db` változó neve csak a `szamol` függvényen belül látható, de maga a változó nem szűnik meg a függvényhívás végén. Megőrzi az értékét a következő hívásig. Ha a függvényt háromszor meghívjuk, akkor a kimenet: `1 2 3`. Ez azért történik, mert a `db` változó csak egyszer jön létre, és a függvényhívások között is megmarad.

A lokális statikus változó tehát átmenetet képez a lokális és a globális változók között: neve lokális, de élettartama globális jellegű.

Jellemzői:

```
deklaráció helye:   globálisan vagy lokálisan is lehet
élettartam:         a program teljes futása alatt
láthatóság:         a deklaráció helyétől függ
memóriaterület:     statikus memóriaterület
kezdőérték:         ha nem adunk meg értéket, automatikusan 0
```

Dinamikus változónak nevezzük azt az adatot, amelynek a memóriáját futás közben foglaljuk le. C-ben ezt leggyakrabban a `malloc`, `calloc` vagy `realloc` függvényekkel végezzük.

A dinamikus változónak általában nincs saját neve úgy, mint egy közönséges változónak. Pointeren keresztül érjük el. A dinamikusan foglalt memória akkor jön létre, amikor meghívjuk a memóriafoglaló függvényt, és addig létezik, amíg fel nem szabadítjuk a `free` függvénnyel. Dinamikus memóriafoglalásra akkor van szükség, amikor a szükséges memória méretét csak futás közben ismerjük.

A dinamikus változók jellemzői:

```
létrehozás helye:   futás közben, malloc/calloc/realloc
                    segítségével
élettartam:         a foglalástól a felszabadításig
láthatóság:         pointereken keresztül érhető el
memóriaterület:     dinamikus memóriaterület, heap
kezdőérték:         malloc esetén nem meghatározott, calloc
                    esetén 0
```

Összefoglalva, a változó nem csupán egy név és egy érték. Egy változóhoz mindig hozzátartozik az is, hogy hol található a memóriában, meddig létezik, és a program mely részei férhetnek hozzá. A globális változó sok helyről elérhető, de emiatt óvatosan kell használni. A lokális változó jól elkülöníti az adatokat, de csak rövid ideig él. A statikus változó megőrzi az értékét a program futása alatt, a dinamikus változó pedig rugalmas memóriahasználatot tesz lehetővé, de a programozónak kell gondoskodnia a felszabadításáról.

A négy felülnézet együtt azt mutatja meg, hogy a C program nem pusztán utasítások, változók és függvények gyűjteménye. A program mögött algoritmus áll;

az algoritmus szerkezetté rendeződik; a szerkezet adatokkal dolgozik; az adatok pedig típusok és változók formájában jelennek meg a memóriában. Ha ezeket az összefüggéseket látjuk, akkor a C nyelv már nem különálló szabályok halmazának tűnik, hanem olyan eszközrendszernek, amellyel tudatosan, áttekinthetően és megbízhatóan lehet programokat építeni. Ez az összkép adja meg az alapot ahhoz, hogy a továbbiakban ne csak használjuk a nyelvet, hanem értsük is a programok működését.

17.2. AlgoRythmics szál: algoritmusok mozgásban

Az AlgoRythmics szál a *keresések és rendezések világába* vezetett be. A célja az volt, hogy az olvasó az algoritmusokat ne csupán C-kódként, hanem lépésekből álló, követhető folyamatként értse meg.

A keresések közül megjelent a *lineáris keresés*, amely sorban vizsgálja az elemeket, valamint a *bináris keresés*, amely rendezett sorozatban felezéssel szűkíti a keresési tartományt.

A rendezések közül a szál bemutatta a *buborékos rendezést*, a *kiválasztásos rendezést*, a *beszúró rendezést*, az *összefésüléses rendezést* és a *gyorsrendezést*. Ezek mind ugyanarra a célra törekednek – az elemek sorba állítására –, de eltérő stratégiával: cserékkel, kiválasztással, beszúrással, részsorozatok összefésülésével vagy rekurzív felosztással.

Az AlgoRythmics-kollekcióban további rendezések is szerepelnek, például a *Shell-rendezés* és a *kupacrendezés*. Ezeket az olvasó önállóan is megtekintheti, és összevetheti a könyvben tárgyalt rendezésekkel: milyen mozgásmintát követnek, hogyan választanak elemeket, hogyan csökkentik a rendezetlenséget, és miben különbözik a ritmusuk a már ismert algoritmusokétól.

Összességében az AlgoRythmics szál azt mutatta meg, hogy a keresés és a rendezés nem pusztán technikai művelet, hanem jól felismerhető gondolkodási minta. A táncos szemléltetés segített láthatóvá tenni az algoritmusok ritmusát, irányát és belső szerkezetét.

17.3. BBC szál: algoritmusok a világ kulisszái mögött

A BBC szál azt mutatta meg, hogy az algoritmusok nemcsak programok belsejében működnek, hanem a mindennapi világ döntéseit, sorrendjeit, kereséseit és optimalizálási folyamatait is alakítják. Segített kilépni a C-kód közvetlen világából, és tágabb összefüggésbe helyezte a könyvben tanult alapfogalmakat.

A szál példái között megjelent a *csoki vagy csili* játék, amely a stratégiai gondolkodást szemléltette; az *euklidészi algoritmus*, amely a padlócsempézés problémáján keresztül mutatta meg a legnagyobb közös osztó keresését; valamint

az *utazóügynök-probléma*, amely az útvonaltervezés és optimalizálás világába vezetett be.

A *Sudoku-ellenőrző* arra világított rá, hogy egy megoldás helyességének ellenőrzése gyakran könnyebb, mint magának a megoldásnak a megtalálása. A *PageRank* a hálózatok és rangsorolás gondolatát hozta közel, míg a *stabil házasság probléma* a párosítási algoritmusok működését szemléltette. Az *ütemezési problémák* azt mutatták meg, hogy sok valós helyzetben a sorrend megválasztása döntő jelentőségű, az *arcfelismerés és képfeldolgozás* példája pedig arra hívta fel a figyelmet, hogy a számítógép a képeket is adatokként, számok és mintázatok formájában kezeli.

Összességében a BBC szál azt tette láthatóvá, hogy a C-ben tanult feltételek, ciklusok, keresések, rendezések, döntések és adatfeldolgozási lépések nagyobb rendszerekben is ugyanilyen gondolkodási mintákra épülnek. Az algoritmus így nem pusztán technikai eszköz, hanem a modern világ egyik szervezőelve.

17.4. Hiúz szál: éleslátás a kód előtt

A Hiúz szál az algoritmikus gondolkodás építéséről szólt: arról, hogyan lehet egy feladat mögött felismerni a rejtett szerkezetet, mielőtt C-kóddá alakítanánk. A hiúz jelképe az éleslátást és figyelmet hordozza: azt a tudatos, analitikus szemléletet, amely a jó programozót jellemzi.

A szál feladatai játékos vagy történetszerű helyzetekből indultak ki, de mindig programozási mintákhoz vezettek. A *csoki vagy csili* a stratégiai gondolkodást és a nyerő lépések felismerését gyakoroltatta. A *prím-teszt* a divízió, oszthatóság és hatékony vizsgálat problémáját hozta közel. A *labirintusból kijutó hiúz* a rekurzív bejárás, az útkeresés és a visszalépés gondolatát szemléltette.

Később a Hiúz szál összetettebb adatszervezési és feldolgozási helyzetekhez vezetett. A *hódok és viperák* feladata sorozatok, pontszámok és feltételes vizsgálatok kezelését kívánta meg. A *házasságok és válások nyilvántartása* már struktúrákban, rendezett tömbökben, keresésben, hibakódokban és állapotváltozásokban gondolkodtatott.

Összességében a Hiúz szál azt mutatta meg, hogy a programozás nem a kódolással kezdődik, hanem a helyzet megértésével. Meg kell látni, milyen adatok tartoznak össze, milyen szabályokat kell követni, hol vannak döntési pontok, mikor kell ismételni, keresni, állapotot módosítani vagy hibát jelezni. A Hiúz szál így hidat épített a logikai feladat és a C program között: előbb a gondolatot élesítette, csak utána következett a kód.

17.5. ECN szál: próbára tett algoritmusok

Az ECN szál azt mutatta meg, hogyan válik a tanult C nyelvi és algoritmikus eszköztár önálló problémamegoldássá. Míg a C szál felépíti az alapokat, az ECN szál versenyfeladatokon keresztül kéri számon: tudjuk-e értelmezni a bemenetet, felismerni a mintát, megválasztani az adatszerkezetet, kezelni a szélső eseteket, majd pontos kimenetet előállítani. A bevezetés szerint az ECN programozóverseny a versenyeztetés révén mélyíti el az algoritmikus gondolkodást.

A szálban megjelent például a *Clock* feladat, amelynél nem elég egy működő ciklust írni: figyelni kell arra is, milyen esetek maradnak kezeletlenül, és miért nem kapna elfogadást egy majdnem jó megoldás.

Az *Open data about train timetable* feladat a versenyprogramozás egyik fontos tanulságát emelte ki: sokszor már a bemenet helyes feldolgozása is a feladat része. Állomásokat, vonatokat, kódokat, neveket és korlátozásokat kell pontosan beolvasni, majd az adatokat összehasonlítható szerkezetbe rendezni.

A *Find the Identical Twins and Triplets* feladat pedig azt mutatta meg, hogyan kapcsolódik össze a struktúra, a dinamikus tömb, a rendezés és az egymás melletti elemek vizsgálata. A feladatban személyek adatait kellett tárolni, személyi kód szerint rendezni, majd az azonos DNS-ű csoportokat hatékonyan felismerni.

Összességében az ECN szál a könyv gyakorlati próbája: nem újabb elszigetelt nyelvi elemeket tanít, hanem azt kérdezi, hogy az olvasó képes-e a tanultakat versenyhelyzetben, pontosan, hatékonyan és önállóan alkalmazni. Itt válik el igazán, hogy a kód nemcsak lefordul-e, hanem minden bemenetre helyes választ ad-e.

►► Zárógondolat

A könyv elején a programozás talán még parancsok, szabályok és különös jelek világának tűnt, a végére azonban kirajzolódhatott mögötte egy másik kép is: a programozás gondolkodásmód. Megtanít pontosan kérdezni, lépésekre bontani, feltételeket felismerni, ismétlődéseket észrevenni, adatokat rendezni, hibákat keresni és a bonyolultat kisebb, kezelhető részekből újra felépíteni.

A C nyelv ebben a könyvben nem önmagáért volt fontos, hanem azért, mert közel engedett a program működéséhez. Láthatóvá tette a változók, típusok, pointerek, függvények, blokkok, modulok és állományok kapcsolatát. Megmutatta, hogy a program nem varázslat, hanem tudatosan megszerkesztett folyamat: adatok mozognak, döntések születnek, ciklusok ismétlődnek, függvények dolgoznak együtt, és mindez végül egy működő megoldássá áll össze.

A különböző szálak ugyanerről beszéltek más-más nézőpontból. A C szál megadta az eszközöket, a Hiúz szál élesítette a látást, az AlgoRythmics szál mozgásba hozta az algoritmusokat, a BBC szál kitágította a látóteret, az ECN szál pedig próbára tette a megszerzett tudást. Együtt azt üzenik, hogy programozni tanulni nemcsak egy nyelv megtanulását jelenti, hanem annak elsajátítását is, hogyan gondolkodjunk rendszerekről, folyamatokról és megoldásokról.

Ha az olvasó a könyv végére nemcsak C programokat tud írni, hanem jobban érti, hogyan kell egy feladatot megközelíteni, hogyan kell a megoldást megtervezni, ellenőrizni és továbbfejleszteni, akkor a könyv elérte célját. A továbblépés innen már nem egyetlen újabb fejezet, hanem önálló út: új feladatok, új nyelvek, új algoritmusok és új felismerések felé. A legfontosabb alap azonban megmarad: a világ sok problémája akkor válik kezelhetővé, ha megtanuljuk meglátni benne az algoritmust.

Irodalomjegyzék

AlgoRythmics.
é. n. AlgoRythmics – Programming Techniques Explained through Folk Dance. YouTube. URL: https://www.youtube.com/algorythmics [2026. július 16.]

SAPIENTIA Erdélyi Magyar Tudományegyetem.
é. n. Sapientia ECN programozói és matematikai csapatverseny. URL: https://ecn.ms.sapientia.ro/ [2026. július 16.]

BEBRAS Community.
é. n. Bebras – International Challenge on Informatics and Computational Thinking. URL: https://www.bebras.org/ [2026. július 16.]

BBC.
2015. The Secret Rules of Modern Living: Algorithms. BBC Four. URL: https://www.bbc.co.uk/programmes/p030s6b3 [2026. július 16.]

CEOI.
2025. Previous CEOIs. Central European Olympiad in Informatics 2025. URL: https://ceoi2025.utcluj.ro/getarticle/15 [2026. július 16.]

DEIK – Debreceni Egyetem Informatikai Kar.
é. n. DEIK Regionális Programozó Csapatverseny. URL: https://inf.unideb.hu/deik-regionalis-programozo-csapatverseny-1 [2026. július 16.]

ELTE Informatikai Kar.
é. n. e-Hód – HÓDítsd meg a biteket! URL: https://e-hod.elte.hu/ [2026. július 16.]

ISO/IEC.
2024. ISO/IEC 9899:2024. Information technology – Programming languages – C. Geneva: International Organization for Standardization. URL: https://www.iso.org/standard/82075.html [2026. július 16.]

KÁTAI Z.
2008. *C nyelv és programozás*. Debreceni Egyetem, Debrecen.

KERNIGHAN, B. W.–RITCHIE, D. M.
1988. *The C Programming Language.* 2nd edition. Englewood Cliffs, Prentice Hall, New Jersey.

KOCH, R.
1997. *The 80/20 Principle: The Secret of Achieving More with Less.* Nicholas Brealey Publishing, London.

RITCHIE, D. M.
1993. The Development of the C Language. *ACM SIGPLAN Notices*, 28(3), 201–208. URL: https://www.nokia.com/bell-labs/about/dennis-m-ritchie/chist.html [2026. július 16.]

WING, J. M.
2006. Computational Thinking. *Communications of the ACM*, 49(3), 33–35. DOI: https://doi.org/10.1145/1118178.1118215 [2026. július 16.]

REZUMAT

Codează în C! – Bazele programării, pas cu pas

Codează în C! este un manual introductiv de programare care urmăreşte formarea treptată a gândirii algoritmice prin intermediul limbajului C. Volumul porneşte de la noţiuni fundamentale, precum variabilele, expresiile, instrucţiunile de decizie şi structurile repetitive, apoi avansează către tablouri, pointeri, funcţii, recursivitate, şiruri de caractere, structuri şi prelucrarea fişierelor. Conţinutul este organizat progresiv, astfel încât cititorul să dobândească mai întâi un nucleu stabil de cunoştinţe, iar ulterior să aprofundeze conceptele în capitole de tip „master".

Caracterul distinctiv al cărţii este dat de structura sa pe mai multe fire tematice. Firul principal prezintă elementele limbajului C, iar secţiunile Râsul (*Hiúz*) dezvoltă rezolvarea logică a problemelor, exemplele AlgoRythmics ilustrează algoritmii prin mişcare şi dans, firul BBC evidenţiază rolul algoritmilor în viaţa reală, iar problemele ECN introduc situaţii de concurs şi soluţii eficiente. Volumul include şi inserţii dedicate inteligenţei artificiale, utilizată ca instrument de reflecţie, verificare şi formulare de întrebări, nu ca substitut al înţelegerii şi al testării programelor.

Cartea se adresează elevilor, studenţilor şi tuturor celor care doresc să înveţe programarea într-o manieră sistematică. Scopul său nu este doar însuşirea sintaxei limbajului C, ci dezvoltarea unei gândiri precise, structurale şi responsabile, transferabile către alte limbaje şi domenii informatice.

ABSTRACT

Code in C! – The Fundamentals of Programming, Step by Step

Code in C! is an introductory programming textbook designed to develop algorithmic thinking step by step through the C programming language. It begins with fundamental concepts such as variables, expressions, decision statements, and loops, and gradually moves toward arrays, pointers, functions, recursion, strings, structures, and file processing. The material is organized progressively: readers first acquire a stable core of essential knowledge and later deepen their understanding in dedicated "master chapters."

A distinctive feature of the book is its multi-threaded pedagogical structure. The main C thread introduces the language and its programming constructs; the Lynx (*Hiúz*) thread focuses on logical and algorithmic problem solving; the AlgoRythmics sections visualize algorithms through movement and dance; the BBC thread connects algorithms to real-life applications; and the ECN thread presents competition-style problems that emphasize correctness, efficiency, and strategic thinking. The book also contains short AI-focused sections in which artificial intelligence is treated as a tool for questioning, reflection, explanation, and verification rather than as a replacement for understanding, compiling, running, and testing programs.

The volume is intended for secondary-school students, university students, and independent learners who seek a structured introduction to programming. Its aim is not merely to teach the syntax of C but to cultivate precise, disciplined, and transferable computational thinking. By linking code, algorithms, visualization, real-world applications, and problem-solving practice, the book presents programming as both a technical skill and a broader intellectual method.

A SZERZŐRŐL

Kátai Zoltán 1968. március 13-án született Nagyváradon. Középiskolai tanulmányait a marosvásárhelyi Bolyai Farkas Elméleti Líceumban végezte 1982–1986 között, egyetemi tanulmányait pedig a Kolozsvári Műszaki Egyetem Automatizálás és Számítógépek Karán 1987–1992 között. 1992–2005 között informatikatanár a marosvásárhelyi Bolyai Farkas Elméleti Líceumban, 2002-től pedig a Sapientia Erdélyi Magyar Tudományegyetem Marosvásárhelyi Kara Matematika–Informatika Tanszékének oktatója és kutatója. Fő kutatási területe a programozási technikák (főleg ezek gráfelméleti háttere), kiemelten a dinamikus programozás. Oktatástudományi vonalon az érzékszervek bevonása a programozásoktatás folyamatába témakörben mélyült el. A szerző kezdeményezte az AlgoRythmics projektet, amelynek keretében született meg a YouTube-on is népszerűvé vált tíz algoritmus-tánckoreográfia. Az AlgoRythmics projekt 2013-ban elnyerte az Informatics Europe által felajánlott Best practice in education díjat.

Scientia Kiadó
400112 Kolozsvár (Cluj-Napoca)
Mátyás király (Matei Corvin) u. 4. sz.
Tel./fax: +40-364-401454
E-mail: scientia@kpi.sapientia.ro
www.scientiakiado.ro

Korrektúra:
Szabó Beáta

Műszaki szerkesztés:
Metaforma Kft.

Tipográfia:
Könczey Elemér

Sorozatborító:
Tipotéka Kft.

Nyomdai munkálatok:
F&F INTERNATIONAL Kft.
Felelős vezető: Ambrus Enikő igazgató

sapientia
tankönyvek

A programozás több mint egy programozási nyelv szabályainak elsajátítása: a logikus, a lépésről lépésre haladó gondolkodás fejlesztése.

A *Kódolj C-vel!* nem egy hagyományos programozási tankönyv. Az alapoktól a mélyebb összefüggésekig építkezik, miközben nem csupán kódolni, hanem algoritmikusan gondolkodni is tanít. Öt egymást kiegészítő szálon vezeti végig az olvasót a C nyelv és az algoritmikus gondolkodás világán. A programozás alapjait a Hiúz logikai kihívásai, az AlgoRythmics táncban megelevenedő algoritmusai, a BBC valós életből merített példái és az ECN programozóversenyek ihlette feladatok teszik élményszerűvé, az AI-cseppek pedig a mesterséges intelligencia tudatos és kritikus használatára ösztönöznek.

ISBN 978-606-975-122-0