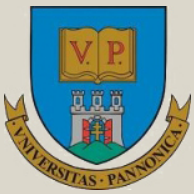
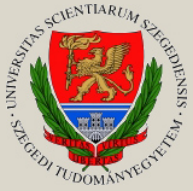




Írta:
SIMON GYULA

A PROGRAMOZÁS ALAPJAI

Egyetemi tananyag

2011

LEKTORÁLTA: Dr. Szeberényi Imre, Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Irányítástechnika és Informatika Tanszék

TÁMOGATÁS:
Készült a TÁMOP-4.1.2-08/1/A-2009-0008 számú, „Tananyagfejlesztés mérnök informatikus, programtervező informatikus és gazdaságinformatikus képzésekhez” című projekt keretében.

ISBN 978-963-279-521-8

KÉSZÜLT: a Typotex Kiadó gondozásában
FELELŐS VEZETŐ: Votisky Zsuzsa
AZ ELEKTRONIKUS KIADÁST ELŐKÉSZÍTETTE: Juhász Lehel

KULCSSZAVAK:
algoritmusok, programok, vezérlési szerkezetek, adatszerkezetek, strukturált program.

ÖSSZEFOGLALÁS:
A jegyzet platform-független módon igyekszik megismertetni a programozás alapjait, a strukturált programozást. Tárgyalja az alapvető programozói háttérismereteket és alapfogalmakat mind a hardver, mind a szoftver oldaláról. Bemutatja az algoritmusok és programok alapvető építőelemeit, valamint a strukturált programok készítésének alapvető szabályait. Az algoritmusok és adatszerkezetek leírására többféle leíró modellt is használ (folyamatábra, Jackson-ábra, reguláris kifejezések és definíciók), valamint a C programozási nyelv segítségével ezekre implementációs példákat is mutat.

Fél a lábam, szemem fél, de én mégse félek,
Falábbal és bekötött szemmel kalóz módra élek.
Pisztolyommal átjutok minden akadályon,
Vállamon meg ücsörög szépen öreg papagájom.
Gryllus Vilmos

Tartalomjegyzék

Előszó

Ebben a jegyzetben a programozás alapjaival foglalkozunk: célunk, hogy a strukturált programozás alapfogalmaival megismertessük az olvasót, megismerkedjünk a programtervezés alapvető eszközeivel és természetesen gyakorlati ismereteket is nyújtsunk. A jegyzet nagyrészt a Pannon Egyetem Műszaki Informatikai Karán (PE-MIK) első éves hallgatóknak meghirdetett *Programozás alapjai* című tárgy anyagára épül.

A programozás alapjaival – amennyire ez lehetséges – nyelv-független módon igyekszünk megismerkedni. Ezért többféle leírással, modellel fogunk dolgozni. De természetesen a programozás lényege az igazi, futtatható kódot írása, tesztelése. Ezért a jegyzetben szükség van egy „igazi” programozási nyelvre is: itt választásunk a C nyelvre esett.

A nyelv választását többek között az indokolta, hogy a PE-MIK képzésében a programozás oktatása a C nyelven alapul, a Programozás alapjai tárgyat követő elméleti és gyakorlati tárgyak zöme is a C nyelvre épül. Jóllehet a C nyelv nem minden elemében ideális a kezdő programozók számára, mégis szilárd alapot ad a további nyelvek elsajátításához. A C nyelv összes lehetőségét most még nem fogjuk kihasználni, csupán a legszükségesebb eszközkészletét fogjuk alkalmazni.

A jegyzetben sárga keretben találjuk a C nyelvre vonatkozó ismereteket. A nyelv szintaktikáját, használatát a számunkra szükséges mélységig itt magyarázzuk el.

Az elméleti ismeretek birtoklásán túl a programozás elsajátításának legfontosabb lépése a gyakorlás. Ennél talán csak egyetlen fontosabb dolog létezik azok számára, akik jó programozók akarnak lenni: még több gyakorlás. Ezért a fejezetek végén az aktuális témakör feldolgozását mindig példákkal segítjük.

A jegyzet elektronikus mellékletben megtalálhatók a szövegben részben vagy egészében kidolgozott programok, valamint néhány gyakorló feladat megoldása is. Az elektronikus mellékletre [szögletes zárójelek között] utalunk.

A jegyzet 1. fejezetében a programozás elsajátításához szükséges háttérismereteket tárgyaljuk: ebben a fejezetben mind a hardverekkel, mint a szoftverekkel kapcsolatos alapfogalmak előkerülnek. A 2. fejezetben tisztázzuk, mik az algoritmusok és programok, majd a 3. fejezetben megismerkedünk a strukturált programok – meglehetősen egyszerű – építőelemeivel és szabályaival. A 4. és 5. fejezetben az algoritmusok, programok leírásának eszközeivel ismerkedünk meg. A 6., 7. és 8. fejezetekben a strukturált programok egyes elemeivel:

a szekvenciális, a szelekciót tartalmazó, valamint az iteratív program- és adatszerkezetekkel, valamint ezek használatával ismerkedünk meg részletesebben. A 9. fejezetben a függvények és eljárások, valamint a változók kezelésének kérdéseivel foglalkozunk. A 10. fejezetben az összetett adatszerkezetekkel foglalkozunk, míg a 11. fejezet a rekurziót és alkalmazási lehetőségeit mutatja be. A 12. fejezet rövid áttekintést ad a programtervezés alapvető módszereiről.

A szöveget helyenként szürke betétek szabdalják. Itt néhány vendég-oktatóval találkozhatunk: Gonosz Géza, Kalóz Karcsi, Féllábú Ferkó és kalóz barátaik igyekeznek színesíteni a száraz szakmai tartalmakat.

Vitorlát fel!

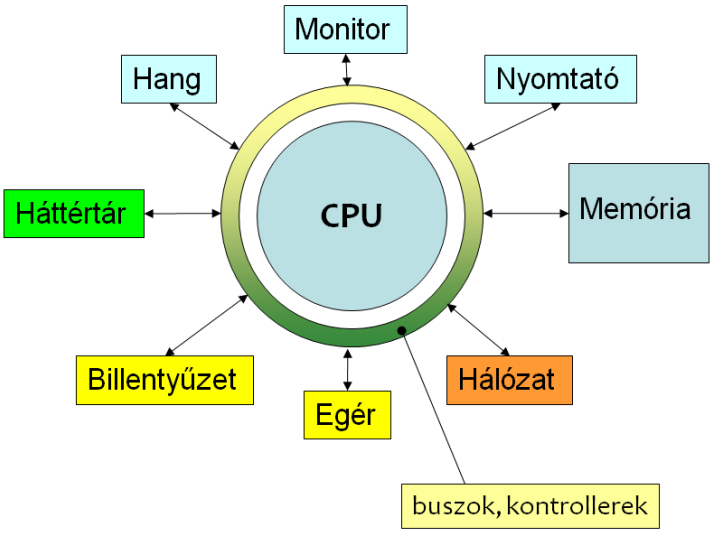
1. fejezet

Bevezetés

Környezetünkben nagyon sok fajta számítógéppel találkozhatunk, néha talán fel sem merül bennünk, hogy az egyáltalán nem számítógép-szerű berendezés is tulajdonképpen egy (vagy akár több) számítógépet is rejthet magában. Ha a számítógép szót halljuk, leginkább egy asztali vagy laptop számítógép képe ugrik be, de telefonjaink, autóink, játékaink is mind-mind tartalmaznak számítógépeket. Igaz, ezeknek néha nincs képernyője és billentyűzete sem, de azért felépítésük és funkciójuk nagyon hasonlít asztali testvéreikére. Ezekben is van CPU, memória, kezelnek perifériákat, és ami a legfontosabb közös vonásuk: programokat hajtanak végre azért, hogy minket segítsenek, szórakoztassanak. Tekintsük tehát át először számítógépeink felépítését.

1.1. A számítógép felépítése

Valamennyi számítógép, legyen az egy irodai asztali számítógép, egy diák laptopja, egy mobiltelefon, az autónk fedélzeti számítógépe, vagy akár a Deep Blue szuper-sakkszámítógép,



1.1. ábra. A számítógép felépítése

tartalmaz néhány nagyon hasonló elemet. A programjainkat minden számítógépben agy (vagy több) CPU hajtja végre, futó programjainkat a memóriában tároljuk, és a programok a külvilággal (velünk, felhasználókkal is) perifériákon keresztül kommunikálnak.

A számítógép főbb elemeit és azok kapcsolatát az 1.1. ábra mutatja, ahol a központi egység, a CPU a memóriával és különféle perifériákkal áll kapcsolatban. Az árán láthatunk jól ismert bemeneti perifériákat (pl. billentyűzet), kimeneti perifériákat (pl. monitor), de a perifériák közé tartoznak a háttértárak és pl. a hálózati kapcsolatot biztosító hálózati csatoló egység is.

1.1.1. A CPU

A CPU rövidítés az angol Central Processing Unit (központi végrehajtó egység) kifejezésből származik. Gyakran használt elnevezés még a *processzor* is. A név jól kifejezi a CPU-k feladatát: ezen a helyen történik a programok végrehajtása. Némi túlzással szokás a CPU-t a számítógép agyának is nevezni; a hasonlat kicsit sántít ugyan, hiszen a CPU nem gondolkodik, viszont villámgyorsan, tévedés nélkül, szolgai módon végrehajtja a programok utasításait. A programok végrehajtása közben a CPU természetesen kapcsolatot tart a memóriával és a perifériákkal. A processzor a memóriából olvassa be a program utasításait, értelmezi ezeket, majd végre is hajtja azokat. Az utasítások hatására változtatásokat hajthat végre környezetén is (a memórián, vagy valamelyik periférián). A processzorutasítások nagyon egyszerűek (pl. két szám összeadása, egy memóriacím olvasása/írása), viszont cserében nagyon gyorsan végrehajthatók: egy kommersz processzor a jegyzet írásakor másodpercenként több milliárd utasítást is képes volt végrehajtani.

A processzorok nagyon bonyolult eszközök, felépítésükkel, működésükkel más tárgyak foglalkoznak. Itt most röviden és a végletekig leegyszerűsítve tárgyaljuk a processzorok felépítésnek alapvető elemeit, amelyek szükségesek ahhoz, hogy a programok végrehajtásának folyamatát megérthessük.

A processzorok számítási képességeit az aritmetikai-logikai egység (az angol Arithmetic Logic Unit névből röviden ALU) biztosítja. Ez az egység képes pl. összeadni két számot, vagy pl. bitenként logikai ÉS műveletet végrehajtani két számon. Az utasításokat a processzor utasítás-beolvasó egysége olvassa be memóriából, majd azokat értelmezve utasítja pl. az ALU-t a megfelelő művelet végrehajtására. A processzorban helyet foglaló regiszterek gyors belső memóriaegységek, melyek pl. az aritmetikai műveletek operandusait és eredményét tárolhatják. A processzor aritmetikai és logikai utasításai általában egy vagy két regiszter között működnek és az eredményeket is valamelyik regiszterben tárolják. (Vannak olyan processzorok is, amelyek képesek a memóriából, a regiszterek megkerülésével is bizonyos műveletek végrehajtani, de nem ez a jellemző.) Tipikus aritmetikai/logikai műveletek lehetnek pl. a következők:

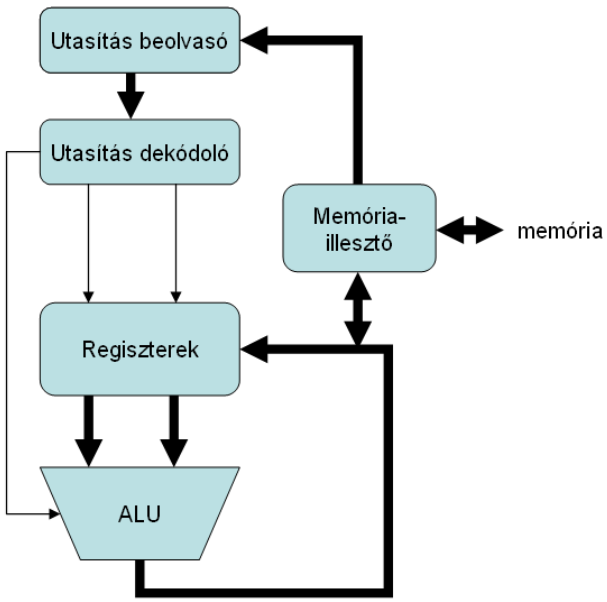
- Add össze az R1 és R5 nevű regiszterek tartalmát és tedd az eredményt az R3 nevű regiszterbe.
- Szorozd össze a R1 és R2 regiszterek tartalmát, az eredményt helyezd az R8 nevű regiszterbe.
- Végezz logikai kizáró vagy (XOR) műveletet az R1 és R2 bitjei között, az eredmény az R3-ban képződjön.

A processzor a fentebb vázolt aritmetikai/logikai utasításokon kívül rendelkeznek mozgató utasításokkal is, amelyek segítségével a memóriából a regiszterekbe lehet adatokat mozgatni, vagy a regiszterekből lehet a memóriába írni. Léteznek ezen kívül regiszterből regiszterbe mozgató, valamint konstansokat a regiszterekbe töltő utasítások is. Ezen utasítások tipikus felhasználási területe a memóriában levő adatok előkészítése a műveletvégzésre, illetve az eredmények tárolása a memóriában. Ilyen műveletek lehetnek pl. a következők:

- Olvasd be a memória 15785-ik rekeszének tartalmát az R1 jelű regiszterbe.
- Mozgasd az R3 tartalmát a memória 62734-ik rekeszébe. (Ezt direkt címzésnek hívjuk, mert a memória címe szerepel az utasításban.)
- Mozgasd az R3 tartalmát arra a memóriacímre, ami az R7-ben van. (Ezt indirekt címzésnek hívjuk, mert maga a memóriacím nem szerepel az utasításban, hanem az egy – az utasításban szereplő – regiszterben található.)
- Mozgasd át az R1 tartalmát az R2-be.
- Töltsd be a 45 számot az R1 regiszterbe.

Az aritmetikai/logikai és a mozgató utasításokon kívül a minden processzor megvalósít bizonyos vezérlési utasításokat, amelyek a program utasításainak végrehajtási sorrendjét befolyásolják. A program utasításait alapesetben egymás után, sorrendben hajtja végre a processzor, kivéve, ha ettől eltérő vezérlő utasítást hajt végre. Tipikus vezérlési utasítások lehetnek pl. a következők:

- Folytasd a program végrehajtását a 358-ik utasításnál. (Ugorj a 358-ik sorra.)
- Ha az előző aritmetikai/logikai utasítás eredménye 0, akkor hagyd ki (ugord át) a következő utasítást.
- Folytasd a program végrehajtását 17 utasítással később. (Ugorj 17 utasítással előre).
- Ugorj vissza 7 utasítást.



1.2. ábra. A CPU felépítése

A processzor működésének illusztrálására definiáljunk egy egyszerű (fiktív) processzort, majd valósítsunk meg vele egy egyszerű feladatot. A processzor számunkra fontos tulajdonságai a következők:

A processzornak 4 regisztere van, ezek: A, B, C, D

Aritmetikai/logikai utasítások:

- A processzor tud aritmetikai és logikai műveleteket végezni két regiszter között, de az eredményt mindig az A regiszterbe teszi. Pl. az ADD B, C utasítás összeadja a B és C regiszterek tartalmát (az eredményt az A regiszterbe teszi), a SUB C, B utasítás pedig kivonja a B regiszter tartalmából a C regiszter tartalmát (majd az eredményt az A regiszterbe teszi).

Mozgató utasítások:

- A processzor a memória tetszőleges címéről be tud olvasni tetszőleges regiszterbe, illetve tetszőleges regiszterből ki tud írni a memória tetszőleges címére.
  Pl. a MOVE A, (165) utasítás kiírja az A regiszter tartalmát a memória 165-ik címére, a MOVE (165), B ugyanazon memóriacímről olvas be az B regiszterbe, míg a MOVE A, (B) utasítás az A regiszter tartalmát mozgatja a memória B regiszter által mutatott címére. (A zárójelek az utasításban arra utalnak, hogy a zárójelben lévő érték – regiszter vagy szám – egy cím.) A MOVE A, B utasítás az A regiszter tartalmát mozgatja a B regiszterbe, míg a MOVE 23, A utasítás 23-at tölt az A regiszterbe (nem pedig a 23-ik cím tartalmát, hiszen itt nincs zárójel).

Vezérlő utasítások

- A processzor a vezérlést tetszőleges sorra át tudja adni (ugrás), pl. a JUMP 34 utasítás a 34-ik programcímen folytatja a program végrehajtást.
- Az elágazást egy feltételes ugrással lehet megvalósítani. Pl. JUMPNZ 34 utasítás a 34-ik programsoron folytatja a végrehajtást, ha az előző aritmetika vagy logikai utasítás eredménye nem 0 volt (itt az utasítás rövid nevében (ún. mnemonikjában) szereplő NZ karaktersorozat a „nem zéró”-ra utal).

A fenti utasítások természetesen ezen a fiktív processzoron léteznek csak, de a valódi processzorok utasításkészlete is hasonló elemekből áll. Most ezen utasítások segítségével valósítsuk meg a következő feladatot.

1.1. Példa: A memóriában annak 20-ik címétől kezdve 10 darab pozitív szám van elhelyezve, minden rekeszben egy. Adjuk össze a számokat és az eredményt írjuk a memória 30-ik címére.

A megvalósítást a következőképpen végezzük: az összeget a D regiszterben tároljuk majd el, ennek tartalmát kezdetben nullára állítjuk. Ehhez hozzáadjuk egyenként a memória 20. és 29. címe között lévő számokat. Az összeadáshoz bemozgatjuk az aktuális adatot a C regiszterbe, amelynek címét a B regiszterben tároljuk.

Az egyszerűség kedvéért a program kezdődjön a memória 0-ik címén és minden utasítás egy memóriacellát foglaljon el. A következő program fiktív processzorunkon az összeadási feladatot oldja meg:

Cím	Utasítás	Megjegyzés
0	MOVE 0, D	; a D regiszterben tároljuk az összeget, ez kezdetben legyen 0
1	MOVE 20, B	; a B regiszterben tároljuk az adatok címét, az elsőé 20
2	MOVE (B), C	; beolvassuk az aktuális adatot a C regiszterbe
3	ADD C, D	; összeadjuk az aktuális adatot az eddigi összeggel
4	MOVE A, D	; eltároljuk az eredményt a D regiszterbe
5	MOVE A, 1	; az A regiszterbe betöltünk egyet a következő összeadáshoz
6	ADD A, B	; a B regiszterben levő címet megnöveljük 1-el
7	MOVE A, B	; az eredményt visszatöltjük a B regiszterbe
8	MOVE 30, A	; az A regiszterbe betöltjük a utolsó adat utáni címet (ez éppen 30)
9	SUB A, B	; majd kivonjuk ebből a B regiszterben lévő címet
10	JMPNZ 2	; ha nem 0 volt, akkor folytatjuk a következő adattal (a 2-ik utasítás-címtől)
11	MOVE D, (30)	; ha az eredmény 0, akkor végeztünk, a D-ben lévő eredményt eltároljuk

A program a memória 0-ik címétől a 11-ig bezárólag foglal helyet. A fenti programlistában az utasításokat az emberek számára is érthető és olvasmányos mnemonikjaikkal jelöltük, de a számítógép memóriájában természetesen nem ezek, hanem az utasítások gépi kódú megfelelői foglalnak helyet. Erre majd látunk konkrét példát is a 1.2.3. fejezetben.

Látható, hogy ehhez az egyszerű feladathoz is viszonylag sok utasításra volt szükség és a feladat megvalósítása nem volt triviális, köszönhetően a furcsa utasításkészletnek. Jóllehet ez már emberi fogyasztásra szánt (mnemonikos) megjelenítés, a programra rápillantva nem egyszerű megmondani, mit is csinál. Persze, aki sokat programoz alacsony szinten, annak egy ilyen utasításkészlet természetes. Akinek pedig a fenti kód első olvasásra nem sokkal érthetőbb, mint egy egyiptomi hieroglifa, az se keseredjen el: mi a továbbiakban majd magasabb szintű programozási nyelveket fogunk használni. De jó, ha észben tartjuk, hogy minden program ilyen primitív gépi utasítások sorozatává alakul át, mielőtt számítógépünk végrehajtaná azt. És persze értékeljük annak szépségét is, hogy magas szintű utasításokat használhatunk a primitív gépi utasítások helyett...

1.1.2. A memória

A memória adatok és programok tárolására szolgál a számítógépben. Egy korszerű architektúrájú számítógépben számtalan helyen találkozhatunk memóriával, de mi most csak a számunkra fontos elemmel, az operatív tárral foglalkozunk, a memória szót ezen értelemben fogjuk használni. (Más, pl. cache memóriáknak is fontos szerepe van a hatékony, gyors működés biztosításánál, de ezek nélkül is működhetnek számítógépek, ráadásul a programozás szempontjából ezek mintha nem is léteznének, ezért mi most ezeket nem tárgyaljuk.)

Az operatív tárban foglalnak helyet az éppen futó programok és a program által használt adatok. Az adatmemória tárolja a bemeneti és kimeneti adatokat, a program futása közben használt átmeneti adatokat; programjaink az adatmemóriát futás közben olvassák és írják is. A programmemória tartalmazza a futó programot; ez is írható és olvasható memória: a CPU

utasítás-beolvasó egysége a program futtatása közben a programmemóriából olvassa be a program utasításait. A programmemória írását általában a program betöltésére korlátozzuk, a program futása közben nem célszerű a programot felülírni (és szerencsére a legtöbb operációs rendszer ezt nem is engedi meg).

Vannak számítógépek, ahol a program és adattárolás feladatát élesen elkülönítik, vagyis külön adat- és programmemóriát használnak (ráadásul még egynél több adatmemória is előfordulhat). Ezeket a gépeket Harvard architektúrájú gépeknek nevezzük. A mindennapi életben gyakoribb architektúrák a Neumann architektúrák, ahol egyetlen memóriát használunk, amely tárolhat egyszerre adatokat és programokat is. A legtöbb személyi számítógép Neumann architektúrájú.

A mai korszerű számítógépekben megszokhattuk, hogy sok programot tudunk egyszerre (párhuzamosan) futtatni. Ezt kényelmesen az ún. virtuális memóriakezelés segítségével lehet megvalósítani. Ez azt jelenti, hogy a számítógépünkön futó valamennyi program úgy látja, hogy egyedül ő birtokolja az egész memóriát, sőt a program akár több memóriát is felhasználhat, mint amennyi fizikailag jelen van a számítógépben. A virtuális memória kezeléséről az operációs rendszer gondoskodik, ami a programozó számára jó (sőt remek!) hír, hiszen így a memóriakezelés nehéz kérdéseivel nem kell foglalkoznia, minden programot úgy készíthet el, mintha a program egyedül futna a számítógépen, ami látszólag korlátlan memóriával rendelkezik.

A memória felfogható rekeszek sorozatának, ahol minden rekeszbe egy adatot tárolunk. Minden rekesznek van egy sorszáma: ha a memória N rekeszből áll, akkor az első rekesz sorszáma a 0, a következő rekeszé az 1, míg az utolsó rekesz sorszáma $N-1$ lesz. A rekeszek sorszámát gyakrabban a memória címének nevezzük.

A rekeszek mérete (szélessége) határozza meg, hogy mekkora adat tárolható benne. Ha a rekesz 8 bites (1 bájtos), akkor 2^8=256-féle érték tárolható benne, míg egy 16 bites (2 bájtos) rekesz 2^{16}=65536 különböző értéket tartalmazhat. Általában a memóriák szélessége a bájt egész számú többszöröse. A mai számítógépekben használt memóriák többségének szélessége 8 és 128 bit között változik.

1.1.3. A perifériák

A perifériák biztosítják, hogy végrehajtás alatt álló programjaink képesek legyenek a külvilággal is kapcsolatot tartani, pl. bemenő adatokat beolvasni vagy az eredményt kiírni. Az egyik legfontosabb periféria a háttértár. A háttértár egy nem felejtő memória (ellentétben az operatív tárban alkalmazott memóriával), így a háttértárra írt információk akkor is megőrződnek, ha a gépünket kikapcsoljuk, vagy éppen laptopunkból kifogy az akkumulátor. A háttértárak tipikusan sokkal nagyobb kapacitásúak, mint az operatív tár, így lényegesen több adat tárolható benne. A háttértárak azonban lényegesen lassabbak is, mit az operatív tár. Tipikus háttértárak a merevlemezek, a szilárdtest meghajtók (SSD meghajtók), vagy a CD és DVD alapú tárolók. A háttértár szerepe tehát az, hogy programjainkat és adatainkat arra az időre is tárolja, amikor azok nem az operatív tárban vannak.

A háttértáron kívül számos perifériával találkozhatunk a számítógépünkön. Tipikus bemeneti perifériák pl. a billentyűzet vagy az egér, kimeneti perifériák a monitor vagy a nyomtató, ki-bemeneti perifériák például a hangkártyák vagy a hálózati csatoló kártyák.

A perifériák gyakran maguk is bonyolult berendezések, amelyek akár saját beépített kis számítógéppel is rendelkezhetnek. A perifériák kezelése nem egyszerű feladat, ezért ennek terhét az operációs rendszer és különféle meghajtó programok veszik le a vállunkról, így programjainkból kényelmesen tudjuk ezeket kezelni, függetlenül attól, hogy éppen milyen típusú és gyártmányú perifériát használunk.

1.2. A programok

A számítógépes program a számítógép számára értelmezhető utasítások sorozata, melyek egy adott feladat megoldására szolgálnak. Egy számítógépes rendszerben számos programmal találkozhatunk, ha figyelmesek vagyunk, de a legtöbb felhasználó – szerencsére – tudomást sem szerez a gépén futó programok többségéről. A felhasználó általában az ún. felhasználói programok miatt használja számítógépét (ilyenek pl. a szövegszerkesztők, böngészők, médialejátszók, stb.), de esetenként az operációs rendszer szolgáltatásait is igénybe veszi (pl. programokat elindít). Az alacsony szintű eszközkezelő programokkal azonban szinte soha nem kell foglalkoznia, ha egy jól bekonfigurált rendszert használ.

1.2.1. A programok kezelése, futtatása

Egy személyi számítógépen programjainkat a (nem felejtő) háttértáron tároljuk. A program indításkor töltődik az operatív tárba, ahonnan az utasításokat a processzor végrehajtja. A futás befejezése után a program törlődik az operatív tárból (de megmarad a háttértáron).

A programok általában telepítéssel kerülnek a számítógépünkre. A telepítés egy egyszerű programnál lehet pusztán annyi, hogy a háttértárunkra másoljuk a programot és máris futtathatjuk. Bonyolultabb programok a telepítés során egyéb beállításokat is végezhetnek. Léteznek olyan programok is, amiket a gyártó telepít fel a számítógépre és a felhasználó ezeket már csak használja (pl. a személyi számítógépek BIOS-a vagy az egyszerűbb mobiltelefonok szoftvere is ilyen).

A programok indítása többféleképpen történhet. Egyes programok a számítógép bekapcsolása után automatikusan elindulnak és állandóan futnak (ilyen pl. egy autó fedélzeti számítógépén futó program, vagy akár a személyi számítógépünk operációs rendszere is). Más programokat a felhasználók indítanak el az operációs rendszer segítségével (pl. kiválasztjuk a programot a start menüből, vagy kettőt kattintunk a program ikonjára, vagy akár parancssorba begépeljük a program nevét). Végül programokat más programok is elindíthatnak (pl. egy böngésző elindít egy médialejátszó alkalmazást).

Ha a számítógépes rendszerünk rendelkezik operációs rendszerrel, akkor az általunk készített programok könnyű és kényelmes kezeléséről, azok biztonságos futtatásáról (más programokkal együtt is) az operációs rendszer gondoskodik. Az operációs rendszer segítségével indul el a program (pl. kettős kattintás után), az operációs rendszer gondoskodik arról, hogy programunk elegendő processzoridőhöz jusson más futó alkalmazások mellett, vigyáz arra, hogy esetlegesen hibás programunk más programokban ne okozhasson kárt (és persze a mi programunkat is megvédi más hibás alkalmazásoktól). Mindezen szolgáltatások igénybevétele azonban teljesen automatikus, ezek nem igényelnek a felhasználó oldaláról semmilyen tevékenységet.

1.2.2. Programok készítése

Ha programot írunk, akkor a számítógép nyelvén, valamilyen programozási nyelv szabályai szerint fogalmazzuk meg azon tennivalókat, amelyeket a számítógéppel végre szeretnénk hajtatni. Mint ahogy az emberi nyelvek, a számítógépes nyelvek is különbözők: egyes nyelvek bonyolultabbak, nehezen tanulhatók meg, de árnyaltabb megfogalmazást tesznek lehetővé rövid mondatokban is (pl. ilyen a francia nyelv), más nyelvek egyszerű szerkezetűek, rövid idő alatt elsajátíthatók, de esetleg hosszadalmasabb magyarázkodás szükséges egy elvontabb gondolat kifejezéséhez (ilyen pl. az eszperantó). Egyes nyelvek kiválóan alkalmasak pl. filozófiai eszmefuttatásokra, míg mások inkább az orrszarvúvadászat részleteinek leírásában jeleskednek. Így van ez a számítógépes nyelvek esetén is: vannak gépközeli (alacsony szintű) és bonyolultabb (magas szintű) programozási nyelvek; vannak általános célú nyelvek, de vannak egyes feladatokra specializálódott nyelvek is (pl. gazdasági számításokra, folyamatirányításra, vagy játékok készítésére).

Az emberi nyelvek tanulásánál közismert jelenség, hogy aki már beszél néhány nyelvet, az a következőt már sokkal gyorsabban sajátítja el. Ez különösen igaz a programozási nyelvekre: a programozás egy gondolkodási forma, ennek megjelenítése egy adott programozási nyelven már csak ezen gondolkodás kivetülése. Ha valamilyen programozási nyelven már jól tudunk programozni, akkor nagyon valószínű, hogy egy új nyelv elsajátítása sem okozhat gondot.

1.2.3. Alacsony és magas szintű nyelvek

A számítógép processzorának nagyon egyszerű utasításkészlete van, ezt nevezzük gépi kódnak. Egy elképzelt processzoron már láttunk egy példát egyszerű gépi kódú programra, most álljon itt egy valódi processzorra írt program, ami a Hello World szöveget írja ki a képernyőre:

```
section .text
    global _start

_start:

        mov     edx,len
        mov     ecx,msg
        mov     ebx,1
        mov     eax,4
        int     0x80

        mov     eax,1
        int     0x80

section .data

msg     db      'Hello, world!',0xa
len     equ     $ - msg
```

Ezt a nyelvet assembly nyelvnek nevezzük, ez a gyakorlatban használt legalacsonyabb szintű nyelv. Az assembly programot az assembler nevű fordító program alakítja át valódi gépi utasításokká. A fenti Hello world program így néz ki gépi kódban:

```
ba66 000e 0000 b966 0024 0000 bb66 0001 0000 b866 0004 0000
80cd b866 0001 0000 80cd 0000 6548 6c6c 2c6f 7720 726f 646c
0a21
```

A fordítás során az egyes utasításokat kölcsönösen egyértelműen lehet a gépi kódú reprezentációhoz rendelni, ezért természetesen az assembly nyelv mindig egy adott processzorhoz (vagy kompatibilis processzorcsaládhoz) van rendelve. A megírt program ráadásul csak az adott számítógépes környezetben fut csak. A fenti példa egy Intel Pentium kompatibilis processzorra íródott (ezen processzor utasításait használja), de használ Linux rendszerhívásokat is, tehát ez a program kizárólag egy Linux operációs rendszert futtató PC-n fog működni. Nem tudjuk futtatni a programot sem egy Windows alapú PC-n, sem egy Linux alapú okostelefonon.

Az ember számára értelmezhetetlen gépi kódhoz képest az assembly nyelv jól olvasható, áttekinthető. De az assembly nyelv egyszerű szerkezete és utasításkészlete nehezen teszi lehetővé bonyolultabb programok írását. Assemblyben gyakran írunk eszközmeghajtókat, de egy szövegszerkesztő vagy adatbázis-kezelő alkalmazáshoz magasabb szintű nyelveket használunk.

Egy programozási nyelvet annál magasabb szintűnek tekintünk, minél közelebb áll a nyelv formalizmusa az emberi gondolkodáshoz. Más megközelítésben annál magasabb szintű egy nyelv, minél összetettebb, bonyolultabb utasításokat lehet benne leírni. Egy magas szintű programozási nyelven könnyebb az algoritmusokat megfogalmazni, könnyebb a programban a hibákat megtalálni, a programot a későbbiekben karban tartani. Minél magasabb szintű azonban egy nyelv, annál távolabb áll a gépi nyelvtől, ezért annál nehezebb hatékony (gyors, kisméretű kódot produkáló) fordítót vagy értelmezőt készíteni hozzá.

Az alábbi példák néhány gyakran használt magas programozási nyelven mutatják be a Hello World program szerkezetét:

C:

```
#include <stdio.h>
int main(void){
	printf("Hello, World!");
	return 0;
}
```

Java:

```
public class HelloWorld {
	public static void main(String[] args) {
		System.out.println("Hello, World!");
	}
}
```

Perl:

```
print "Hello, World!\n";
```

Aki nem tud a fenti nyelveken programozni, az is kis angol nyelvismerettel valószínűleg megérti a print szóból, hogy itt kiírásról van szó. Aki pedig már látott valamilyen programozási nyelvet, az az átadott paraméterből azt is kitalálja, hogy itt a „Hello, World” karaktersorozatot íratjuk ki. Az egyes nyelvek tartalmaznak még több-kevesebb, a feladat szempontjából „felesleges” kódrészletet (a Perl a legkevesebbet), de ezek rövid „nyelvtan-tanulás” után érthetővé válnak.

Eddig a programozási nyelvek kapcsán értekeztünk alacsonyabb és magasabb szintekről. Vizsgáljuk meg most ezt a gondolatkört egy meglehetősen hétköznapi példán keresztül:

A rettegett kalózvezér, Morc Misi elásott egy kincses ládát egy lakatlan szigeten. Egy térképen bejelöli a kincs lelőhelyét. Vannak azonban kissé képzetlen kalózai is, akik nem tudnak térképet olvasni. Ezért egy papírra feljegyzést készíttet a kincskeresés miként jéről. Segédje és bizalmasa, Féllábú Ferkó a térkép alapján a következő listát írja fel:

1. Szállj ki a teknős alakú sziklánál
2. Haladj 50 métert a part mentén kelet felé
3. Fordulj a sziget belseje felé
4. Haladj 30 métert a pálmafák irányába
5. Fordulj nyugat felé 30 fokkal
6. Haladj 10 métert előre
7. Áss 5 méter mélyre

A lista elkészítésekor Féllábú Ferkónak fontos szempont volt, hogy csak olyan elemi utasításokat alkalmazzon, amit az egyszerű kalózok is megértenek és végre is tudnak hajtani: pl. haladj *x* métert előre, fordulj el *x* fokkal, stb. A kincskeresést végrehajtó kalóznak nem kell gondolkodnia, csak pontosan egymás után végre kell hajtania az utasításokat.

Mivel a kalózok gondolkodása a tetemes rumfogyasztás miatt elég csökevényes, ezért Féllábú Ferkónak ügyelnie kell arra, hogy az utasításokat pontosan, megfelelő sorrendben írja le. Nem követhet el olyan hibát, hogy a „haladj” helyett azt írja, hogy „adj halat”, mert a buta kalóz nem tudja kitalálni Ferkó szándékát. Ugyancsak a kincs elvesztéséhez vezet, ha Ferkó 5 méter mély ásás helyett véletlenül 5 láb mély ásást ír elő. Ha véletlenül felcserél két utasítást, akkor a kincskereső kalóz rossz helyen fog kutatni (nem mindegy, hogy „Fordulj nyugat felé 30 fokkal” és utána „Haladj 10 métert előre”, vagy „Haladj 10 métert előre” és utána „Fordulj nyugat felé 30 fokkal”).

Analógiák (csak annyira sántítanak, mint Féllábú Ferkó):

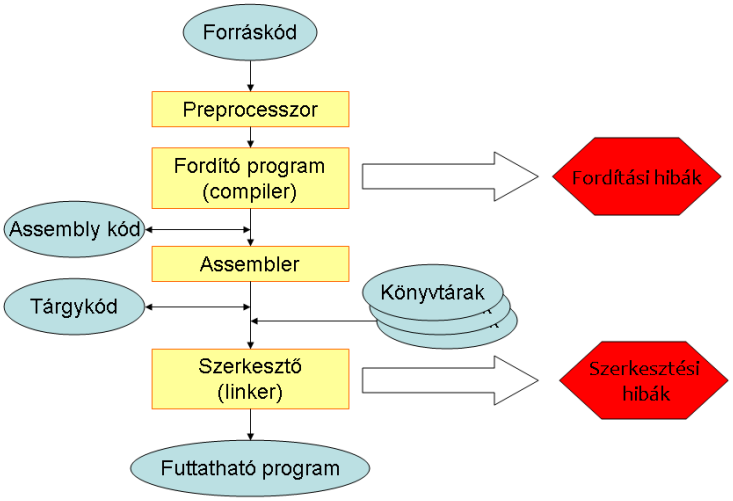
- A papírra írt utasítássorozat egy számítógépes programra hasonlít.
- Féllábú Ferkó programja a papíron, a számítógépes programok a memóriában tárolódnak (amikor futtatjuk).
- Féllábú Ferkó programját egy egyszerű kalóz hajtja végre, a számítógépes programot a CPU.
- Az egyszerű kalóz felel meg a CPU-nak (egyik sem gondolkodik, csak parancsot hajt vége). Itt azonban meg jegyezni, hogy míg a CPU mindig gyorsan és pontosan hajtja végre a parancsot (nem téved), addig a kalóz az elfogyasztott rum mennyiségétől függő sebességgel és minőségben hajtja végre az utasításokat.
- Féllábú Ferkó utasításai a CPU gépi nyelvének utasításainak felelnek meg.

- Ferkó által a lista készítésekor elkövetett hibák a programban szintaktikai vagy szemantikai hibáknak felelnek meg. A szintaktikai hibák kiszűrése egyszerűbb (akárki láthatja, hogy az „adj halat” értelmetlenség), míg a szemantika hibáké (pl. a felcserélt utasítások, méter-láb összekeverése) csak úgy lehetséges, ha Ferkó összeveti a listát a térképpel és ellenőrzi, hogy helyes-e a leírt utasítássorozat.

1.2.4. Fordítás és értelmezés

A magas szintű programnyelven leírt utasításokat a processzor nem tudja értelmezni, ezeket előbb a processzor gépi nyelvére át kell alakítani. A gépi nyelvű reprezentációt már a processzor képes futtatni. Ha az átalakítás a futtatás előtt történik, akkor ezt fordításnak nevezzük, az átalakítást végző segédprogram pedig a *fordító* (vagy *compiler*). Fordítást általában a program megírása után végezzük, az elkészült gépi kódú állományt pedig ezek után bármikor futtathatjuk. Mivel a fordítást csak egyszer végezzük el, mégpedig általában jóval a futtatás előtt, így nem kritikus, hogy mennyi ideig tart a fordítási folyamat. Ezért gyakran a fordító programok inkább lassabban dolgoznak, hogy minél hatékonyabb gépi kódú programot tudjanak generálni.

1.2. Példa: A C nyelvű forráskódot le kell fordítani, mielőtt futtatni tudjuk a programot. A programunkat C nyelven írjuk, pl. a hello.c fájlba. Ezután a forráskódot lefordítjuk (pl. Linux alatt a
`gcc -o hello hello.c` paranccsal), aminek eredményeképp előáll a futtatható állomány



1.3. ábra. A C forráskód fordítási folyamata

(példánkban `hello` néven). A fordító a fordítási folyamat során a forráskódunkat több lépésben alakítja át futtatható állománnyá: először a *preprocesszor* egyszerű előfeldolgozást végez a forráskódon (pl. a `#define` direktívákat behelyettesíti), majd a fordító a C forráskódból assembly programot készít. (Az assembly programot meg is nézhetjük, ha a fordítást
a `gcc -S hello.c` paranccsal végezzük; ekkor pl. a `hello.s` állományban megtaláljuk az assembly kódot.) Az assembly állományból az assembler készít tárgykódot. (A tárgykód példánkban a `gcc -c hello.c` paranccsal állítható elő, de a `hello.o` állomány már emberi szemnek nem túl olvasmányos.) Végül a tárgykódból a felhasznált könyvtári elemekkel a *szerkesztő* (vagy *linker*) állítja elő a végleges futtatható állományt (példánkban `hello` néven). A fordítás vagy szerkesztés közben talált hibákról természetesen hibaüzeneteket kapunk. A fordítás folyamatát az 1.3. ábra illusztrálja.

A fordítást végezhetjük a példa szerint parancssorból, vagy használhatunk kényelmes integrált fejlesztői környezeteket is, ahol a fordítást egyetlen egérkattintással indíthatjuk. Az integrált környezetek további nagy előnye, hogy a hibakereséséhez hatékony és kényelmes eszközöket adnak.

Ha az forráskód futtatható formára történő átalakítását futtatás közben végezzük, akkor *értelmezés*ről beszélünk, az átalakítást végző programot pedig *értelmező program*nak, vagy *interpreter*nek nevezzük. Az interpreter a magas szintű programkódot a futtatás közben ellenőrzi, értelmezi és hajtja végre. . Az interpretált programok általában lassabban futnak, mint a fordított programok, hiszen az értelmezés idejével megnő a futtatás ideje.

1.3. Példa: A perl nyelven írt forráskódot nem kell lefordítani, mert azt a perl interpreter segítségével azonnal futtatni tudjuk. Pl. Linux alatt a `hello.perl` nevű forrásprogram a `perl hello.perl` paranccsal futtatható. Itt a `perl` parancs magát a perl interpretert indítja, amely futás közben értelmezi a `hello.perl` állományban talált forráskódot és azonnal végre is hajtja azt.

A fordítás nagy előnye, hogy a futtatható állomány a program készítésekor elkészül, a futtatás már gyorsan történik. A fordítás közben azonban gépfüggő kód keletkezik, így az egyik platformra fordított futtatható állományt egy másik platformon nem lehet futtatni (pl. a Linux alatt fordított `hello program` nem fog futni Windows alatt). Ezért a fordított programok erősen platformfüggők, ha hordozni kívánjuk programjainkat, akkor minden platformra külön le kell fordítani azokat (az adott platformra írott fordítóprogrammal). Az értelmezett nyelveknél nincs ilyen probléma, ott magát a forráskódot hordozhatjuk, ami valamennyi platformon – ahol az adott nyelvre elkészítették az értelmező programot – ugyanúgy fog futni. Pl. a `hello.perl` program Windows és Linux alatt is ugyanazt az eredményt adja (természetesen Windows alatt más perl parancsértelmező program fut, mint Linux alatt). Az értelmezett nyelveken írt programok azonban általában lényegesen lassabban futnak, mint a fordítottak. Hogy a fordítás nagy sebességét a hordozhatósággal ötvözni lehessen, létrejöttek közbülső megoldások, amelyek tartalmaznak mind fordítási, mint értelmezési fázisokat. Ezen nyelveken írt forráskódot nem a gépi nyelvre, hanem egy közbülső kódra fordítjuk le. A fordítás során megtörténik a szintaktikai ellenőrzés, s így előáll egy platformfüggetlen közbülső kód. Ezt a kódot minden platformon egy értelmező dolgozza fel, ami értelmezi, vagy egyes megvalósításokban futás közben állítja elő a platformfüggő gépi nyelvű kódot. Ilyen típusú nyelv pl. a Java.

1.4. Példa: A java nyelvű `HelloWorld.java` nevű forráskódunkat először bájtkódra fordítjuk le (a Java esetén bájtkódnak nevezzük a platformfüggetlen közbülső kódot). A fordítást pl. a `javac HelloWorld.java` paranccsal végezhetjük el. A fordítás eredménye a `HelloWorld.class` nevű állományban található. Ezt a bájtkódot akár Windows, akár Linux alatt a `java HelloWorld` paranccsal futtathatjuk.

Morc Misi reggel napiparancsot hirdet alvezéreinek:

„Gonosz Géza, hozzátok el a szigetről az elrejtett kincset.”

„Kalóz Karcsi, raboljátok el és hozzátok ide Lady Gagát, koncertet adunk a születésnapomon.”

„Vak Viki, ha élve hazaértek, etesd meg a papagájt.”

...

A kalózvezérek csapatukkal elindulnak végrehajtani a parancsokat. Gonosz Géza pl. a következő utasításokat kiáltja emberinek:

Horgonyt fel!

Vitorlát fel!

Irány dél-délkelet!

Mikor a szigethez érnek, a következő utasításokat harsogja:

Vitorlát le!

Horgonyt le!

Végül kiküld egy csónakot a teknős alakú szikla felé Féllábú Ferkó utasításaival:

Szállj ki a teknős alakú sziklánál

Haladj 50 métert a part mentén kelet felé

...

A kincs sikeres kiásása után pedig hazahajózik:

Horgonyt fel!

Vitorlát fel!

Irány észak-északnyugat!

...

A napiparancsok lényegesen bonyolultabbak, mint a „menj x métert előre” típusú egyszerű utasítások, ezeket az egyszerű kalóz nem tudná végrehajtani. Amikor Gonosz Géza csapatával elindul a kincsért, Géza lebontja a bonyolult parancsot egyszerű utasításokká, amit az emberei is megértenek: pl. „Horgonyt fel!”, vagy „Szállj ki a teknős alakú sziklánál”. Ebben a példában a következő analógiákat fedezhetjük fel:

- Morc Misi napiparancsa egy magas szintű nyelven megírt programnak felel meg.
- A napiparancsbeli parancsok a magas szintű nyelv utasításai (ezeket a végrehajtó egység – egyszerű kalóz vagy a CPU – még nem tudja értelmezni)
- Gonosz Géza (és a többi alvezér) egy parancsértelmezőnek felel meg. Ő fordítja le a magas szintű utasításokat a végrehajtók számára is érthető és végrehajtható alacsony szintű utasítások sorozatára.

Ha kalózos példánkban további analógiákat keresünk, Féllábú Ferkó listáját tekinthetjük a magas szintű „Ássátok ki a kincset” utasítás gépi nyelvre lefordított változatának. Itt a fordító Féllábú Ferkó volt, aki képes volt a bonyolult parancsot egyszerű utasításokká alakítani. Persze lehetőség van másféle értelmezésre is: Féllábú Ferkót tekinthetjük egy assembly nyelvet használó programozónak is, aki maga készíti el a programot alacsony – gépi szintű – utasítások segítségével.

1.3. Háttérismeretek

1.3.1. Számrendszerek, számábrázolás

A mai digitális eszközeink kétállapotú logikával működnek. Ez részben fizikai, részben történelmi-technológiai okokra vezethető vissza: könnyű olyan rendszereket készíteni, amelyekben az elemek két jól definiált állapot valamelyikében vannak. A legalapvetőbb kétállapotú eszköz a kapcsoló, illetve az ehhez tartozó lámpa: a kapcsoló vagy fel van kapcsolva, vagy le van kapcsolva, köztes állapot nem lehetséges (a villanykapcsolókban pl. egy egyszerű rugós szerkezet biztosítja, hogy ne lehessen a kapcsolót „félúton” megállítani, az elengedett kapcsoló az mindig az egyik végállapotba billen). Az digitális elektronikus eszközök szintén kétállapotú, bináris elemekből épülnek fel, ahol a két állapotnak pl. egy az alacsony vagy magas feszültség felel meg. Az ilyen rendszerekben tárolt információ is kétállapotú, ahol az egység a *bit*. Egy bit értéke lehet 1 vagy 0, igaz vagy hamis. A bináris rendszerekben természetszerűen a kettes számrendszeren alapuló számábrázolást használjuk.

1.5. Példa:

$11_2 = 1*2 + 1*1 = 3$

$1001_2 = 1*8 + 0*4 + 0*2+ 1*1 = 9$

$110010101_2 = 1*256 + 1*128 + 0*64 + 0*32 + 1*16 + 0*8* 1*4 + 0*2 + 1*1 = 405$

Nagyobb mennyiségek leírására gyakran használunk a bit helyett *bájt*okat. A bájt nyolc bitből áll, ezért egy bájton 2^8=256 érték ábrázolható (0, 1, 2, 3, ..., 253, 254, 255).

Mivel számok kettes számrendszerbeli ábrázolása hosszú, a gyakorlatban nem szoktuk azt mondani, hogy a bájt értéke 10010101, hanem helyette vagy a tízes számrendszerbeli (decimális) értékét közöljük (149), vagy helyette a hexadecimális számábrázolást hívjuk segítségül, ami a 16-os számrendszeren alapul.

Egy bájtot két négyes csoportra bonthatunk (felső négy bit, alsó négy bit). Ezen bitnégyeseken összesen 2^4=16 számérték ábrázolható (0, 1, 2, ..., 15). Sajnos a 9-nél nagyobb értékekhez a decimális ábrázolásban már két számjegyre van szükség, ezért ezt a hat számot helyettesítsük az A, B, C, D, E és F karakterekkel, a következők szerint:

Decimális:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
hexadecimális:	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Így egy bájtot két hexadecimális karakterrel lehet jellemezni, pl. 6A, vagy 44. A 6A-ról azonnal sejtjük, hogy hexadecimális szám, de a 44 karaktersorozatról nem lehet eldönteni, hogy az a negyvennégy decimális számot jelenti, vagy a négy-négy hexadecimális számot. Ezért a hexadecimális számokat megegyezés szerint valamilyen módon jelöljük, pl. írhatunk mögé egy h karaktert (6Ah, 44h) , vagy írhatunk elé egy 0x karaktersorozatot (0x6A, 0x44).

Természetesen nem csak bájtokat ábrázolhatunk hexadecimális formában, hanem bármilyen számot is. A hexadecimális számábrázolás nem más, mint a szám 16-os számrendszerbeli reprezentációja.

1.6. Példa:

3Eh = 0x3E = 3*16 + 14 = 62

E4CAh = 0xE4CA = $14*16^3 + 4*16^2 + 12*16 + 10 = 58570$

25h = 0x25 = 2 * 16 + 5 = 37 (≠ 25!)

FFh = 0xFF = 15*16 +15 = 255

FFFFh = = 0xFFFF = $15*16^3 + 15*16^2 + 15*16 + 15 = 65535$

1.3.2. Prefixumok

A mindennapi életben gyakran használunk váltószámokat (prefixumokat). Általában a boltban egy kilogramm kenyeret kérünk és nem ezer gramm kenyeret. Két város távolságát kilométerben adjuk meg, a csavarok méretét milliméterben, míg a fény hullámhosszát nanométerben. Az SI prefixumok jelentését a legtöbb ember alapfokú tanulmányaiból ismeri. Azt is tudjuk, hogy az SI prefixumok a nagyságrendeket az ezer (10^3) hatványai szerint módosítják. Tehát a milli jelentése egy ezred, a kilo ezerszeres, míg a mega egymilliószoros szorzót jelent.

Az informatika területén sajnálatos módon ugyanezeket a prefixumokat kicsit más értelemben használjuk, ami bizony félreértésekre adhat okot. Itt a tíz (illetve az ezer) hatványai helyett a kettő (illetve a 2^{10}=1024) hatványait használjuk.

Az informatikai prefixumok az SI prefixumokhoz közeli arányokat jeleznek (pl. a kilo 1000 helyett 1024-et), de az eltérés a nagyobb prefixumok felé egyre nagyobb. A félreértések elkerülésére az IEC (International Electrotechnical Commission) 1998-ban javasolta, hogy az SI prefixumokat kizárólag decimális alapú értelemben használjuk. Bináris értelemben (2 hatványainak jelölésére) új elnevezést és jelölést javasolt. Ennek lényege, hogy a jelölésben az SI prefixumhoz egy i betűt illesztünk, a kiejtésben pedig az SI prefixum első két betűjét kiegészítjük a bi (binary) végződéssel. Ezt a javaslatot az ISO/IEC szabványügyi testülete 2000-ben elfogadta, majd 2005-ben és 2008-ban megerősítette és kiegészítette. A javaslatot a Magyar Szabványügyi Testület 2007-ben MSZ EN 60027-2 szám alatt honosította.

SI (decimális)				IEC (bináris)			
Prefixum		Értéke		Prefixum		Értéke	
jele	neve			jele	neve		
k,K	kilo	10^3	1.000	Ki	kibi	2^{10}	1.024
M	mega	10^6	1.000.000	Mi	mebi	2^{20}	1.048.576
G	giga	10^9	1.000.000.000	Gi	gibi	2^{30}	1.073.741.824
T	tera	10^{12}	1.000.000.000.000	Ti	tebi	2^{40}	1.099.511.627.776
P	peta	10^{15}	1.000.000.000.000.000	Pi	pebi	2^{50}	1.125.899.906.842.624
E	exa	10^{18}	1.000.000.000.000.000.000	Ei	exbi	2^{60}	1.152.921.504.606.846.976

Megjegyzés: az informatikában nem használunk egynél kisebb prefixumokat, tehát nincs millibájt, de van terabájt.

1.7. Példa:

Ha egy 4 GiB-os (4 gibi bájtos) memória modulunk van, akkor abban 4096 megabájt, azaz 4.194.304 kilobájt, azaz 4.294.967.296 bájt információt tudunk tárolni. Ezzel szemben 4 TB-os háttértárolón 4000 gigabájt, azaz 4.000.000 megabájt, azaz 4.000.000.000 kilobájt, azaz 4.000.000.000.000 bájt információt tudunk tárolni.

1.3.3. Az ASCII kódolás

Programjainkban gyakran előfordul, hogy karaktereket olvasunk be, dolgozunk fel, vagy jelenítünk meg. Ezért érdemes megismerkedni a karakterek számítógépes ábrázolásával, kódolásával.

Az egyik leggyakrabban használt karakterkódolási szabvány az ASCII (**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange), ahol karaktereket egy bájton kódolják. A szabványos ASCII karakterek az alsó hét bitet foglalják el, az F1. függelékben található táblázat szerint.

A táblázatban a sötétebb mezők az ún. vezérlő karaktereket jelölik, amelyek nagy része már csak történeti jelentőséggel bír (pl. a 8-es kódú BEL karakter,aminek kiírása egy csengő hangot szólaltat meg), de még néhányat aktívan alkalmazunk (pl. ilyen a 9-es kódú tabulátor vagy a 10-es (0xA) kódú soremelés).

1.8. Példa:

Az „A” karaktert a 0x41 = 65 szám jelöli.

A kis „a” karakter kódja a 0x61 = 97.

A „b” kódja a 0x62 = 98.

Az egyes számjegy „1” kódja a 0x31 = 49.

Figyelem: az „1” karakter nem azonos az 1 számmal! Az 1 egész számot pl. a számítógép így kódolhatja egy bájton: 0000 0001, míg az „1” karakter ASCII kódja 0011 0001.

Az ASCII kódolás csak igen korlátozott mennyiségű karakter kódolására elegendő. Sajnos az ASCII kód nem tartalmazza a magyar ábécé összes betűjét, nem is beszélve a nem latin betűket használó nyelvek karaktereiről. Így számos más kódolási szabványt is kidolgoztak, aminek segítségével (hosszabb kódokat) alkalmazva ezen karaktereket is ábrázolni lehet.

Az ASCII táblázatot (vagy egyéb kódokat) természetesen nem kell fejből tudni, sőt egy magas szintű programozási nyelvnél egyáltalán nem is kell használni (Pl. egyes nyelvekben pl. a Q karakterre úgy hivatkozhatunk, hogy ’Q’, a tabulátor karakterre pedig így: ’\t’). De mindenképpen hasznos, ha észben tartjuk azt, hogy a karaktereket is egy bitsorozatként tárolja a számítógép. Minél alacsonyabb szintű nyelvet használunk, annál inkább tisztában kell lennünk azzal, hogy milyen kódolást (pl. hány bájt hosszút) alkalmazunk karaktereink kódolására.

Feladatok:

1.1. Töltsük ki a következő táblázat hiányzó elemeit a példa szerint. Minden sorban ugyanazon szám különböző számrendszerbeli ábrázolásai szerepeljenek.

Decimális	Bináris	Hexadecimális
143	10001111	8F
17		
149		
43690		
61937		
	1011000	
	11110011100	
	10011100001111	
	1101010000110001	
		FA
		1234
		F0C1
		ABCDE

1.2. Próbáljuk meg a fejezetben szereplő C, Java és Perl nyelvű Hello World programok valamelyikét (esetleg mindet) futtatni.

1.3. Fejtsük meg az alábbi ASCII-kódolt szöveget: 69h, 70h, 61h, 66h, 61h, 69h, 20h, 70h, 61h, 70h

1.4. Egy háttértáron filmeket tárolunk átlagosan 700MiB hosszú állományokban. A háttértár mérete 20TiB. Hány filmet tudunk rajta eltárolni?

2. fejezet

Algoritmusok és programok

Az algoritmus szó valószínűleg Muhammad ibn Musa Al-Khwarizmi perzsa matematikus (és polihisztor) nevéből származik, de a legismertebb algoritmus mégis Eukleidész nevéhez kötődik. Ez a híres módszer alkalmas két természetes szám legnagyobb közös osztójának meghatározására. Az algoritmus egy lehetséges megfogalmazása a következő:

ALG2.1. Legyen x és y két szám úgy, hogy $x \geq y > 0$. Keressük a legnagyobb közös osztójukat.

1. Legyen r az x és y maradékos osztásának maradéka.
2. x helyébe tegyük az y-t, y helyére az r-t.
3. Ha $y = 0$, akkor x a legnagyobb közös osztó, különben ismételjük az 1. lépéstől.

Egy másik, ugyancsak ismert algoritmus a másodfokú egyenlet valós gyökeinek meghatározására szolgál:

ALG2.2. Keressük az $ax^2 + bx + c = 0$ egyenlet valós megoldásait.

1. Az alábbi képletekbe helyettesítsük be az egyenlet együtthatóit:
   $x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$, $x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$.
2. Ha a négyzetgyök alatti kifejezés nem negatív, akkor az egyenlet gyökei x_1 és x_2, egyébként az egyenletnek nincs valós gyöke.

Egy harmadik algoritmus a koca-motorosoknak segít a hiba javításában.

ALG2.3. Nem indul a motor. Keressük meg és javítsuk ki a hibát.

1. Ellenőrizd a benzintankot. Ha nincs benne benzin, akkor töltsd tele.
2. Ha még mindig nem indul, akkor ellenőrizd az akkumulátort. Ha nincs megfelelően feltöltve, akkor töltsd fel.
3. Ha még mindig nem indul, ellenőrizd a gyertyát. Ha szükséges, cseréld ki.
4. Ha még mindig nem indul, vidd el szerelőhöz.

Az algoritmusnak számos lehetséges meghatározása, definíciója létezik, mi most alkalmazzuk a következő definíciót:

Az algoritmus véges számú utasításból áll, amelyek egy feladat végrehajtására szolgálnak. Az algoritmus egy bemenő állapotból indulva az utasításokat jól definiált sorrendben végre-

hajtva a végállapotba kerül (megáll). Az algoritmustól megköveteljük, hogy minden utasítás végrehajtása után a következő utasítás egyértelműen meghatározott és korlátozás nélkül végrehajtható legyen, valamint azt is, hogy véges számú lépésben befejeződjön.

Vizsgáljuk meg, hogy a fenti definíciót kielégíti-e ALG2.1. Az ALG2.1 utasításai az 1, 2 és 3 jelű lépésekben vannak leírva (tehát valóban véges számú – pontosan három – utasításból áll). Ezen utasítások sorrendje is egyértelműen meg van határozva (az 1. és 2. lépéseket ismételjük, amíg meg nem találjuk a legnagyobb közös osztót). Ezen utasítások egymás után mindig végrehajthatók, amíg az algoritmus meg nem áll (vagyis *y* nem nulla). Az algoritmus garantáltan véges lépésben befejeződik (hiszen *y* értéke minden lépésben csökken, vagyis *y* véges lépésben szükségszerűen 0 lesz, ami az algoritmus megállásához vezet).

Hasonlóan a fenti definícióminden kritériumának megfelelnek ALG2.2 és ALG2.3 is, tehát ezek is algoritmusoknak tekinthetők.

Vizsgáljuk meg, hogy a következő leírás algoritmus-e:

ALG2.4. Keressük az $ax^2+bx+c=0$ egyenlet valós megoldásait.

1. Oldd meg az $ax^2+bx+c=0$ egyenletet

ALG2.4 hasonló problémát old meg, mint ALG2.2, de ALG2.4-ben csak egyetlen – nagyon tömör – utasítás található („Old meg az $ax^2+bx+c=0$ egyenletet”). Ha ezen utasítás a definíció szerint „egyértelműen meghatározott”, illetve „korlátozás nélkül végrehajtható”, akkor a definíció minden további követelménye is triviálisan teljesül, tehát ALG2.4 is algoritmus. Ha az egyértelmű meghatározottság feltétele nem teljesül, akkor ALG2.4 nem algoritmus.

Az utasítás egyértelmű meghatározottsága és korlátozás nélküli végrehajthatósága azt jelenti, hogy az utasítás végrehajtója pontosan érti és végre is tudja hajtani az utasítást. Egy középiskolát végzett ember érti, mit jelent az „Oldd meg az $ax^2+bx+c=0$ egyenletet” utasítás, be tudja helyettesíteni az együtthatókat a megoldó képletbe, tudja, hogy mikor van valós gyök és mikor nincs, anélkül, hogy ezt részletesebben elmagyaráznánk neki: számára ALG2.4 algoritmus. Egy általános iskolás diák azonban még nem ért a másodfokú egyenletekhez, így számára ALG2.4 nem értelmezhető, neki ez ilyen formában nem algoritmus. Azonban ha részletesebben elmagyarázzuk neki, mit is jelent a másodfokú egyenlet megoldása, számára érthető módon kifejtjük neki a bonyolult utasítást – mint azt ALG2.2-ben tettük – akkor egy zsebszámológép segítségével ő is meg tudja oldani a feladatot. A fenti tanulságok alapján definiáljuk az elemi és összetett tevékenységek fogalmát:

Elemi tevékenységnek nevezzük azt a legbonyolultabb tevékenységet, amely a végrehajtó számára közvetlenül érthető és egyértelműen végrehajtható anélkül, hogy azt további részekre bontanánk. Az összetett tevékenységek ezzel szemben több elemi tevékenységből állnak.

Korábbi kalózos példánkban a napiparancs a következő utasításokból állt:

„Gonosz Géza, hozzátok el a szigetről az elrejtett kincset.”

„Kalóz Karcsi, raboljátok el és hozzátok ide Lady Gagát, koncertet adunk a születésnapomon.”

„Vak Viki, ha élve hazaértek, etesd meg a papagájt.”

A napiparancsban minden utasítás egy elemi utasítás volt az alvezérek (Gonosz Géza, Kalóz Karcsi és Vak Viki) számára. Ők ezeket a parancsokat értették és végre tudták hajtani anélkül, hogy Morc Misinek tovább kellett volna magyaráznia. Gonosz Géza tudta, hogyan

kell a szigetig elhajózni és ott kiásni a kincset. Kalóz Karcsi is tisztában volt azzal, hogy ki az a Lady Gaga és hogyan kell elrabolni őt. Vak Viki pedig szakértője a papagájetetésnek, így neki sem okoz gondot a parancs értelmezése és végrehajtása.

Az egyszerű matrózok számára ezen utasítások nem elemi utasítások, tehát ezeket tovább kell részletezni számukra. Így Gonosz Géza is egyszerűbb parancsok sorozatára bontotta a tevékenységet („Vitorlát fel”, vagy „Szállj ki a teknős alakú sziklánál”). Ezen parancsok már elemi parancsok a matrózok számára, így azokat pontosan végre is tudják hajtani. Az alvezérek intelligensebb emberek, ezért jobban is fizetik őket, mit a matrózokat. A vezér ezért el is várja tőlük, hogy legyenek képesek bonyolultabb, *magasabb szintű* feladatokat megoldani.

Eddigi algoritmusainkat különféle személyek hajtották végre. Ha az algoritmus végrehajtását egy gépre bízzuk, akkor elérkezünk a program fogalmához:

A számítógépes program olyan algoritmus, amelynek minden elemi tevékenysége a számítógép számára értelmezhető.

Hasonlóan a kalóz alvezérekhez és egyszerű matrózokhoz, programjaink végrehajtó eszközei is lehetnek intelligensebbek vagy egyszerűbbek. Amint láttuk, a számítógép végrehajtó egysége számára értelmezhető legelemibb utasítás a gépi kódú utasítás. Egy ilyen utasítás assembly nyelvi megfelelője pl. a következő utasítás:

U1: `mov ebx,1`

Ez az utasítás egy x86 architektúrájú processzoron az `ebx` nevű regiszterbe mozgat 1-et. Ez egy nagyon alacsony szintű utasítás, ilyenekből elég fáradságos megírni a másodfokú egyenlet megoldó képletét. Az assembly nyelv *alacsony szintű programozási nyelv*.

A következő utasítás már magasabb szintű elemekből építkezik, a leírt utasítás hasonlít ahhoz, amit matematikaórán írnánk (az sqrt jelenti a gyökvonást):

U2: `x1=(-3+sqrt(3^2-4*2*1))/(2*2);`

A gép a parancs végrehajtása után az `x1` változóba betölti az egyenlet egyik gyökét (`x1=-0.5`). Ilyen típusú utasításokat szinte minden általános célú programozási nyelven megadhatunk (pl. C-ben is). Ezeket a nyelveket *magas szintű programozási nyelv*eknek nevezzük. Persze a magas szintű nyelvek között is vannak különbségek, egyes nyelvek bizonyos problémákra igen magas szintű utasításokat tartalmazhatnak. Pl. a következő MATLAB parancsban csak a megoldandó egyenletet közöljük a géppel, a megoldó képlettel már nem kell bajlódnunk:

U3: `x=solve('2*x^2+3*x+1=0')`

A gép válasza itt már tartalmazza mindkét megoldást, sőt a -0.5 lebegőpontos formátum helyett az eredményt a pontos tört-formátumban kaptuk meg.

```
x =
  -1/2
    -1
```

Nyilvánvalóan az U3 példában jutottunk el a legmagasabb absztrakciós szintre, ahol az adott feladat – a másodfokú egyenlet megoldása – közvetlen megfogalmazására alkalmas nagyon magas szintű parancs is rendelkezésre állt. Ezen a nyelven nem kellett a megoldás miként jén gondolkodni, csak a feladatot kellett az adott nyelven megfogalmazni (solve – „Oldd meg a következő egyenletet”). Itt a program írása közbeni tévedések lehetősége kicsi, a program jól áttekinthető és egyértelmű. Az U2 példában egy általános matematikai appará-

tus állt rendelkezésünkre, ahol nekünk kellett a megoldás mikéntjét – a megoldó képletet – leprogramozni, de még mindig elég magas absztrakciós szinten: voltak változóink, a matematikai műveletek jelölése ismerős volt. A kód olvasása közben viszonylag hamar rájövünk, hogy mit is csinál a program, de már a programunk bonyolultabb, terjengősebb, a hibák elkövetésére is több lehetőségünk van. A U1 példában az absztrakciós szint nagyon alacsony volt, a megoldás során használt apparátusnak alig volt köze a feladathoz: nincsenek változók, az aritmetikai műveletek nehezen áttekinthetők, a program nem olvasmányos. Ilyen alacsony szintű nyelven nagyon könnyű hibákat véteni és ezen hibák megtalálása is nehézkes.

Tegyük fel, hogy egy számítógépes rendszerben a következő elemi utasítások léteznek:

- `beolvas: X`: beolvassa az X nevű változó értékét
- `kiír: X` kiírja az X nevű változó értékét
- `kiír: „szöveg”`: kiírja a „szöveg” karaktersorozatot
- Aritmetikai műveletek (összeadás, szorzás, gyökvonás, stb.) változók segítségével

Oldjuk meg ezzel a géppel a másodfokú egyenletet (azaz adjunk olyan másodfokú egyenlet megoldására alkalmas algoritmust, aminek elemei műveletei a gép elemi műveleteivel egyeznek meg).

A felhasznált változóink legyenek *a*, *b*, *c*, *D*, x_1 és x_2, amelyeket rendre az egyenlet együtthatóinak, a diszkriminánsnak, valamint a két gyök értékeinek tárolására használunk. Az algoritmus például a következő lehet:

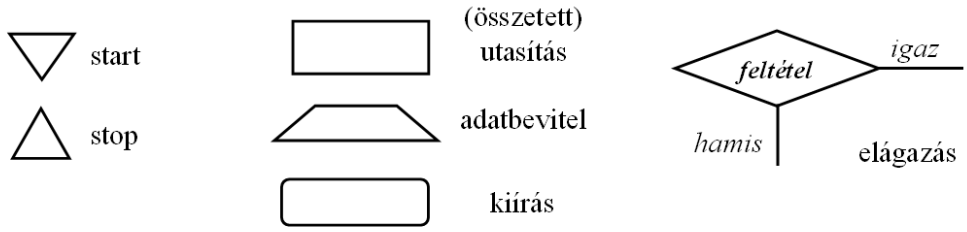
beolvas: *a*
beolvas: *b*
beolvas: *c*
legyen *D* értéke $b^2 - 4ac$
ha *D* negatív akkor
    kiír: „Nincs valós gyök”
különben
    legyen x_1 értéke $\frac{-b+\sqrt{D}}{2a}$, legyen x_2 értéke $\frac{-b-\sqrt{D}}{2a}$
    kiír: „az első gyök” x_1
    kiír: „a második gyök” x_2

Az algoritmus fenti leírása az emberi nyelvhez közeli, mégis formalizált nyelven történt. Pl. a „ha ... akkor ... különben ...” fordulatot változatlan formában találjuk meg a leírásban, míg a beszélt nyelvi „olvasd be *a*-t” parancs helyett a ragozatlan, tömörebb „beolvas: *a*” parancsot alkalmaztuk. Az ilyen típusú leírásokat pszeudo-kódnak nevezzük. A pszeudo-kód további tárgyalására, a nyelvi szabályok részletes ismertetésére és finomítására a 3. fejezetben kerül majd sor.

Algoritmusainkat gyakran szemléletes, grafikus formában is megfogalmazzuk. Az egyik ilyen gyakran használt leírási mód a folyamatábra. A folyamatábra néhány egyszerű elemből – amelyek különféle tevékenységeket szimbolizálnak –, valamint ezen elemeket összekötő nyilakból állnak. A nyilak az elemek egymás utáni végrehajtási sorrendjét mutatják. Minden tevékenységből pontosan egy nyíl vezet a következő végrehajtandó tevékenységhez – kivéve

az elágazást, ahonnan két nyíl is vezet tovább. A folyamatábra végrehajtási szabálya egyszerű: olyan sorrendben kell a tevékenységeket végrehajtani, ahogyan azt a nyilak mutatják.

A folyamatábra elemeit a 2.1. ábra mutatja. A folyamatábra végrehajtása mindig a start szimbólumnál kezdődik és a stop szimbólumnál végződik. A folyamatábrában az utasításokat – amelyek lehetnek összetett utasítások is – téglalapba írjuk. Két speciális utasításnak külön



2.1. ábra. A folyamatábra elemei

szimbóluma van, ezek az adatbevitel (trapéz) és a kiírás (lekerekített sarkú téglalap). A folyamatábrában ezen kívül használhatunk elágazást is (rombusz). Az egyes elemek között a végrehajtási sorrendet nyilak jelzik. A start szimbólumból csak kifelé, míg a stop szimbólumba csak befelé vezet nyíl. Az utasításokba mindig egy nyíl vezet befelé és egy nyíl vezet kifelé. Az elágazásba egy nyíl vezet befelé és két nyíl vezet kifelé: a végrehajtás a feltétel kiértékelése függvényében folytatódhat az egyik vagy másik ágon.

Megjegyzés: a folyamatábrák jelölésére többféle irányzat létezik, ezekben kissé eltérő módon jelölik az egyes tevékenységeket. Természetesen bármelyik jelölés megfelelő; mi most használjuk a 2.1. ábrán látható készletet.

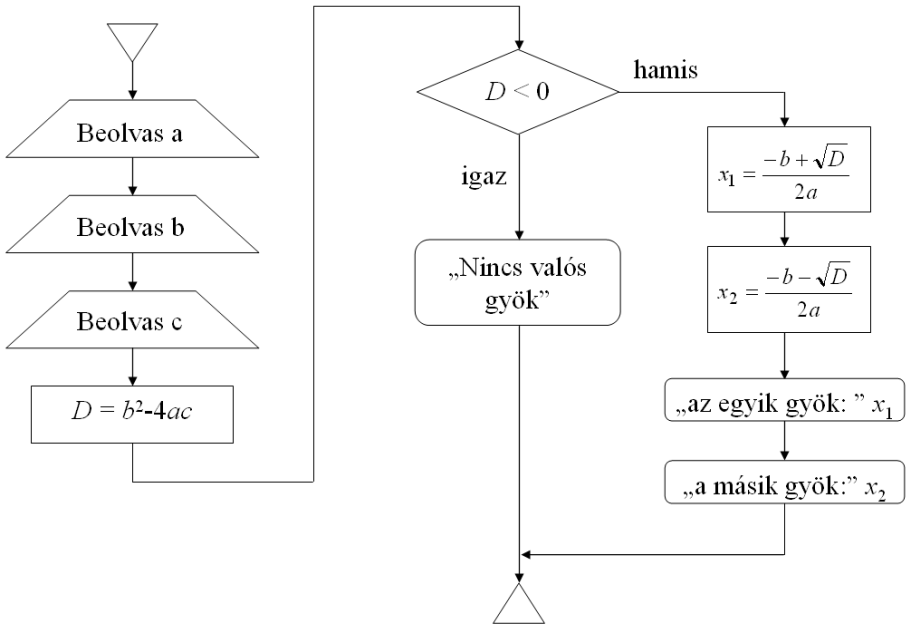
A másodfokú egyenlet megoldását az 2.2. ábrán látható folyamatábra írja le. A program végrehajtása a start szimbólumnál kezdődik, majd a három beolvasó utasítás következik egymás után. Ezek után a diszkrimináns kiszámítás történik meg, majd az elágazásnál folytatódik a végrehajtás. Ha a feltétel ($D < 0$) igaz, akkor program a feltétel szimbólum „igaz” feliratú nyilánál folytatódik az üzenet kiírásával. Ha a feltétel hamis, akkor a végrehajtás a gyökök kiszámításával és a két gyök kiírásával folytatódik. A program mindkét ága a stop szimbólumba torkollik, itt ér véget a program végrehajtása.

A folyamatábrát természetesen nem csak számítógépes programok, hanem bármilyen algoritmus leírására is alkalmazhatjuk. Az ALG2.3 jelű motorjavító algoritmus leírása a 2.3. ábrán látható. A folyamatábra pontosan azokat a tevékenységeket írja, mint az ALG2.3 szöveges (pszeudo-kódos) változat, de a vizuálisabb típusú emberek talán jobban átlátják a tevékenységeket és azok sorrendjét.

Feladatok:

2.1. Írjuk fel a délelőtti tevékenységeinket szövegesen, majd folyamatábrával is. A tevékenységek az ébredéssel kezdődjenek és az ebéddel fejeződjenek be.

Vegyük figyelembe, hogy hétköznapokon mást csinálunk, mint a hétvégén.

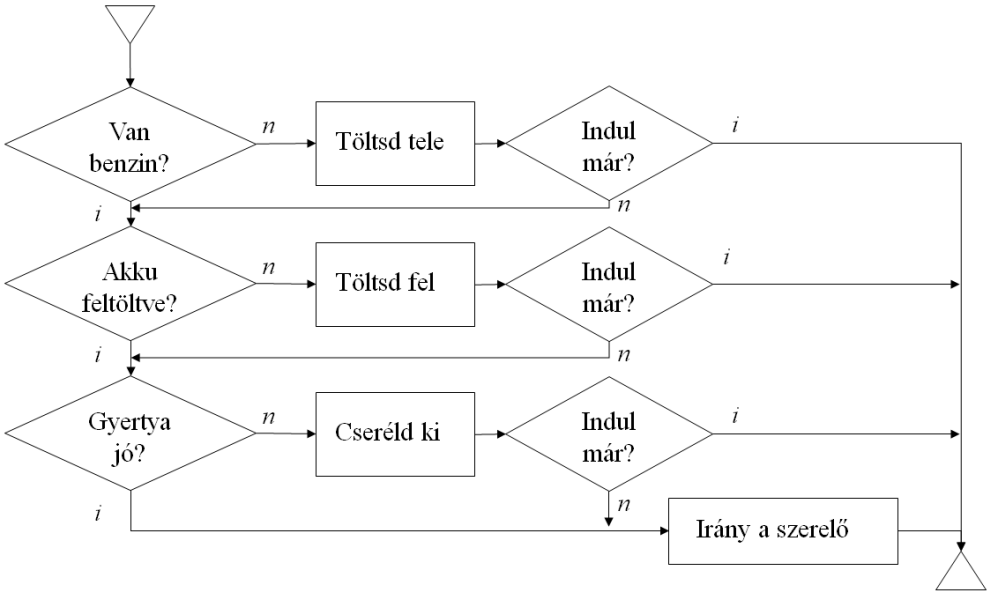


2.2. ábra. A másodfokú egyenlet megoldásának folyamatábrája

2.2. Írjuk le kedvenc receptünk (pl. palacsinta) elkészítési módját folyamatábrával.

Az algoritmus ügyeljen arra is, hogy egyes hozzávalók esetleg nincsenek otthon, ezeket a sütés-főzés előtt be kell szerezni.

A hozzávalókat általában a közeli boltban meg lehet vásárolni, de sajnos néha itt nem minden kapható: ilyenkor el kell menni a távolabbi szupermarketbe is.



2.3. ábra. A motorkerékpár-ápolás kocáknak (ALG2.3) – folyamatábrával

2.3. Írjuk le az öltözködés folyamatát folyamatábra segítségével. Öltözetünk változzon az időjárásnak és az alkalomnak megfelelően. Viseljünk bátran nadrágot és szoknyát (urak kiltet) is.

2.4. Írjuk le az ALG2.1 (két szám legnagyobb közös osztóját kereső) algoritmust folyamatábrával.

Egészítsük ki az algoritmust a két szám (x és y) beolvasásával és az eredmény kiírásával is. Ügyeljünk arra, hogy ALG2.1 feltételezi, hogy $x \geq y$. Ezt ellenőrizzük és ha nem így van, írjunk ki hibaüzenetet.

Egészítsük ki az algoritmust úgy, hogy az $x < y$ esetén is működjön. Tipp: ilyenkor cseréljük meg a két számot beolvasás után. (A megoldást a 3. fejezetben ismertetjük.)

3. fejezet

Alapvető vezérlési szerkezetek és a strukturált program

Algoritmusainkat, programjainkat nem gépek, hanem emberek írják, tartják karban. Ezért a programok készítésénél figyelembe kell venni emberi mivoltunkból eredő korlátainkat. Ez irányba mutató nyilvánvaló törekvés a magas szintű programozási nyelvek használata: az emberek sokkal könnyebben átlátnak olyan típusú leírásokat, amelyek közelebb állnak az emberi gondolkodáshoz, kifejezési formákhoz, mint mondjuk egy gépi kódban leírt számsorozatot.

Az emberi korlátok közül egy fontos tényező korlátozott memóriánk. Egyszerű tesztekkel igazolható, hogy az emberek átlagosan egyszerre körülbelül 7 dolgot tudnak észben tartani. Ennél sokkal több elemű rendszereket már nehezen látunk át. Ez a korlát természetesen a programozásra is igaz: ha egy olyan algoritmust készítünk, amely igen sok alkotórészből áll, akkor ez előbb vagy utóbb áttekinthetetlenné válik, mind a készítője, de különösen más olvasók számára. Célszerű tehát törekedni arra, hogy algoritmusaink viszonylag kevés építőelemből álljanak.

Az emberi gondolkodás erősen sémákon alapul, szeretünk jól ismert elemekből építkezni. A programozási tevékenységek esetében ez azt jelenti, hogy egy jól kezelhető programban (az egyébként kevés számú) részegység kapcsolata jól definiált sémákon alapuljon. Ezen kapcsolati sémákat tevékenységszerkezeteknek nevezzük és összesen három alapvető tevékenységszerkezetet használunk. Ezek a következők:

- sorozat (vagy szekvencia),
- elágazás (vagy szelekció),
- ciklus (vagy iteráció).

3.1. Tevékenységsorozatok

A tevékenységsorozat (szekvencia) a legegyszerűbb tevékenységszerkezet: egyszerűen azt jelenti, hogy különféle tevékenységeket végzünk, mégpedig egyiket a másik után. Szekvenciák előfordulnak a leghétköznapibb helyzetekben: pl. vizet engedek a pohárba, majd belefacsarom egy citrom levét, majd kevés cukorral ízesítem. Egy ilyen – logikailag egységbe tartozó –

tevékenységsorozatot kezelhetünk egyetlen összetett tevékenységként is, az előbbi példánkban pl. hívhatom ezt az összetett tevékenységet limonádékészítésnek is.

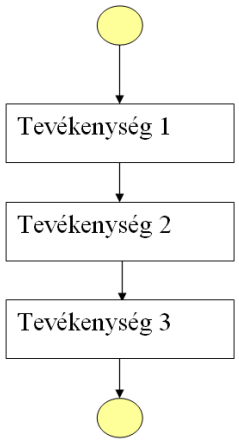
További példák hétköznapi szekvenciákra:

- Hétfő reggel magyar óra van, aztán történelem, utána angol, majd matek következik.
- A húst jól megmossuk, apró darabokra szeleteljük. A hagymát apróra vágjuk, a füstölt szalonnát feldaraboljuk. Felszeletelünk egy paradicsomot és egy paprikát is... (Gulyás recept)
- A pohár száját nedvesítsük be egy citromdarabkával majd mártsuk egy tál sóba. Két rész tequilát, egy rész Triple Sec-et, egy rész lime-levet sok jéggel jól rázzunk össze egy shakerben, majd szűrjük az előkészített pohárba. Díszítsük egy lime-darabbal. (Margarita)
- Elmegyek a boltba. Megveszem a gyümölcsöt, zöldséget, a kenyeret, majd a felvágottat. Hazaviszem az árukat. Megterítek a vacsorához. Jóízűen falatozunk.

Amennyiben tevékenységsorozatot pszeudo-kóddal szeretnénk leírni, akkor egyszerűen a tevékenységeket egymás alá írjuk. Egy három elemű szekvencia esetén pl. így nézhet ki a pszeudo-kód:

```
Tevékenység 1
Tevékenység 2
Tevékenység 3
```

Ha folyamatábrát szeretnénk használni a szekvencia leírására, akkor a tevékenységeket (pl. egymás alá rajzolva, de más topológia is éppen ilyen jó) nyilakkal összekötve ábrázoljuk, mint azt a 3.1. ábra mutatja.



3.1. ábra. Szekvencia jelölése folyamatábrával

A 3.1. ábrán látható folyamatábrának nem része a két kis kör: ezek azt hangsúlyozzák csupán, hogy a szekvenciának, mint összetett vezérlési szerkezetnek egyetlen belépési pontja („bejárata”) és egyetlen kilépési pontja („kijárata”) van, függetlenül attól, hogy a szekvencia belsejében mi történik. Mindig egyetlen ponton kezdjük a szekvencia végrehajtását és mindig egyetlen ponton fejezzük azt be, akármennyi és akármilyen tevékenységet is végzünk el egymás után a kezdet és befejezés között. Ez a tulajdonság nagyon fontos közös tulajdonsága lesz valamennyi vezérlési szerkezetnek: akármilyen is a vezérlési szerkezet, annak mindig egyetlen belépési és egyetlen kilépési pontja lesz.

Fontos megjegyezni, hogy egy tevékenységszerkezetben – így a szekvenciában is – használt tevékenységek lehetnek akár elemi tevékenységek, vagy lehetnek összetett tevékenységek, más tevékenységszerkezetek is.

3.2. Elágazások

Gyakran tevékenységeink valamilyen feltétel kiértékelésének eredményétől függenek. Programjainkban az elágazás (szelekció) tevékenységszerkezet egy feltétel kiértékelésének függvényében alternatív végrehajtási módokat ír le. A szelekció egyik fontos eleme a feltétel, amelyet kiértékelünk a szelekció végrehajtása előtt. Ennek eredményétől függ, hogy a lehetséges tevékenységek közül melyik tevékenységet hajtsuk végre. A szelekció többi eleme a lehetséges végrehajtandó tevékenységek halmaza, amelyből mindig legfeljebb egy hajtódik végre. A hétköznapi életben a választások, döntések mind szelekcióval írhatók le.

Példák a mindennapi életben használt elágazásokra:

- Ha jó meleg idő lesz, rövid nadrágban és pólóban strandra megyek, különben pedig farmert és inget fogok viselni a kertészkedéshez.
- Ha jól sikerül a vizsga, a barátaimmal elmegyünk megünnepelni. (Különben nem megyünk...)
- Hétköznapokon fél hétkor, szombaton nyolc órakor, vasárnap pedig fél kilenckor kelek.
- Ha a pincér búsás borravalót kap, előveszi a legszebb mosolyát.
- Ha a heti lottó nyeremény 100 millió fölött van, a szenvedélyes játékosok még több lottószelvényt vesznek.
- Ha ebéd közben cseng a telefon és valamelyik családtagom hív, akkor felveszem, különben nem.
- Nyáron meleg van, télen hideg van.
- „Ha a világ rigó lenne, / kötényemben ő fütyülne, / éjjel-nappal szépen szólna, / ha a világ rigó volna.”

A legegyszerűbb elágazást pszeudo-kóddal a következőképpen jelölhetjük:

```
ha feltétel akkor
     Tevékenység
elágazás vége
```

Ez az elágazás a feltétel függvényében vagy végrehajtja a tevékenységet, vagy nem. Egy példa ezzel a formalizmussal leírva:

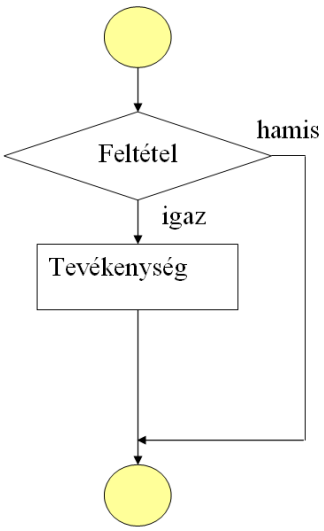
```
ha jól sikerül a vizsga akkor
     a barátaimmal elmegyünk ünnepelni
elágazás vége
```

A fenti jelölésekben a „ha”, „akkor” és „elágazás vége” úgynevezett kulcsszavak, megállapodás szerint ezeket használjuk jelölésrendszerünkben. A következőkben még a kulcsszavak

listáját kicsit bővítjük majd a „különben” kulcsszóval. Azért éppen ezeket a kulcsszavakat választottuk, mert ezek hasonlítanak a beszélt nyelvben használt fordulatokra, ezért jól áttekinthetővé, könnyen olvashatóvá teszik a pszeudo-kódot. Természetesen használhatnánk akár angol kulcsszavakat is (pl. „if”, „then”, „else”, „end”), vagy akármilyen megállapodás szerinti szimbólumokat (pl. kismacit a „ha” helyett, egérkét az „akkor” helyett, stb.), de most maradjunk ennél a logikus, magyar szavakból álló kulcsszókészletnél. A további tevékenységszerkezetek jelölésére is magyar nyelvű kulcsszavakat fogunk használni a pszeudo-kódokban.

Az elágazásban használt feltétel egy logikai kifejezés, amelynek értéke igaz vagy hamis lehet. Helyes feltétel tehát a példában látott „jól sikerül a vizsga”, hiszen egy vagy teljesül, vagy nem. További helyes feltételek lehetnek pl.: szép idő lesz holnap, hétköznap van, szomjas vagyok, az A változó értéke páros, a D változó értéke nagyobb nullánál, A nagyobb B-nél, stb. Nem helyes feltételek a következők: hány éves Zoli (nem eldöntendő kérdés), R négyzet π (ez sem eldöntendő kérdés).

Az elágazások jelölésére a folyamatábrában már találkoztunk egy speciális szimbólummal (rombusz), amely az egyetlen olyan szimbólum a folyamatábrában, amelynek két kimenete van. A folyamatábra elágazás-szimbóluma azonban nem azonos a elágazás tevékenységszerkezettel! A folyamatábra elágazás-szimbóluma csupán egy építőelem lesz az elágazás tevékenységszerkezetben, és ugyanezt a szimbólumot fogjuk alkalmazni majd a ciklusok jelölésére is. Tehát a tevékenységszerkezetet nem maga a szimbólum, hanem a szimbólumból épített struktúra határozza meg. A legegyszerűbb elágazás tevékenységszerkezet jelölése a 3.2. ábrán látható: amennyiben a feltétel igaz, végrehajtjuk a tevékenységet, különben pedig a program végrehajtása a tevékenység kihagyásával folytatódik. Fontos megfigyelni egy nagyon fontos kötöttséget az ábrán: a hamis ág nem akárhova vezet, hanem pontosan az igaz ág befejezése utáni pontra mutat. Más szavakkal: az elágazás tevékenységszerkezet elején a feltétel függvényében a végrehajtás elágazódik, de a tevékenységszerkezet végén a két végrehajtási szál ismét összefonódik. Az elágazás tevékenységszerkezet kívülről tehát ismét a már ismert – minden tevékenységszerkezetre jellemző – képet mutatja: egy belépési pontja és egy kilépési pontja van.



3.2. ábra. A legegyszerűbb elágazás tevékenységszerkezet jelölése folyamatábrával

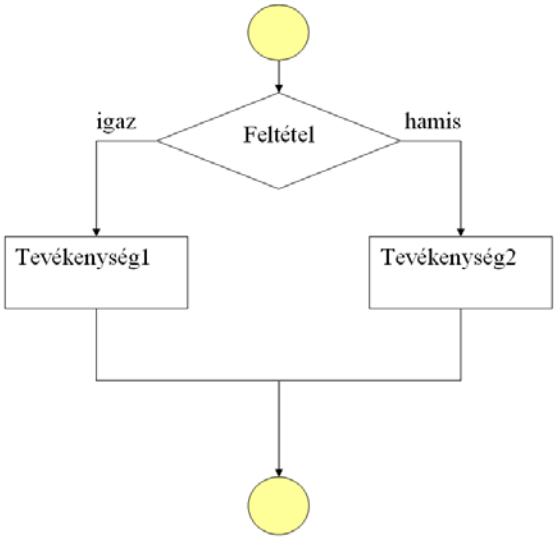
Egy kicsit összetettebb elágazás két tevékenység közül választ:

```
ha feltétel akkor
Tevékenység1
különben
Tevékenység2
elágazás vége
```

Ebben az elágazástípusban még mindig csak egy feltétel van, de már két tevékenység közül választhatunk (az előző esetben az egyik tevékenység a semmittevés volt). Gyakorlat példáink közül pl. ilyen a következő:

```
ha jó meleg idő lesz akkor
rövid nadrágot és pólót veszek fel
strandra megyek
különben
farmert és inget veszek fel
kertészkedek
elágazás vége
```

A fenti példában láthatjuk, hogy az elágazás tevékenységei lehetnek összetett tevékenységek is: jelen esetben minden ágon két egymást közvető egyszerűbb tevékenység található – vagyis egy-egy szekvencia. Az elágazásra is igaz, hogy az egyes ágak tevékenységei tetszőlegesen bonyolult tevékenységszerkezetek is lehetnek. A fenti példában szekvenciákat látunk, de ezek lehetnek akár további elágazások, ciklusok is, illetve ezekből összeállított bármilyen tevékenységszerkezet. Az ilyen elágazás folyamatábrás jelölését a 3.3. ábrán láthatjuk. Itt is jól megfigyelhető a tevékenységszerkezet belsejében szétvált vezérlés (vagy Tevékenység1, vagy Tevékenység2), ami a egyetlen szállá fut össze a tevékenységszerkezet végén, hogy az egy belépési-, egy kilépési pont szerkezet megmaradjon.



3.3. ábra. Kétágú elágazás jelölése folyamatábrával

Amennyiben az elágazás során több lehetőség közül választhatunk, a következőképpen írhatjuk le pszeudo-kóddal:

```
ha feltétel1 akkor
Tevékenység1
különben ha feltétel2 akkor
Tevékenység2
különben ha feltétel3 akkor
Tevékenység3
különben
Tevékenység4
elágazás vége
```

Egy valós példa ilyen bonyolultabb elágazásra a következő

```
ha hétköznap van akkor
ébresztő 6:30-kor
különben ha szombat van akkor
ébresztő 8:00-kor
különben
ébresztő 8:30-kor
elágazás vége
```

Megjegyzések:

1. A többágú elágazásra bevezetett jelölés nem feltétlenül szükséges, pl. kétágú elágazásokból is mindent meg lehet oldani. Az áttekinthetőbb kód miatt azonban mégis alkalmazzuk a többágú elágazásokra szolgáló egyszerűbb jelölést. Nézzük meg, hogy nézne ki egy többágú elágazás egymásba ágyazott kétágú elágazásokkal megoldva:

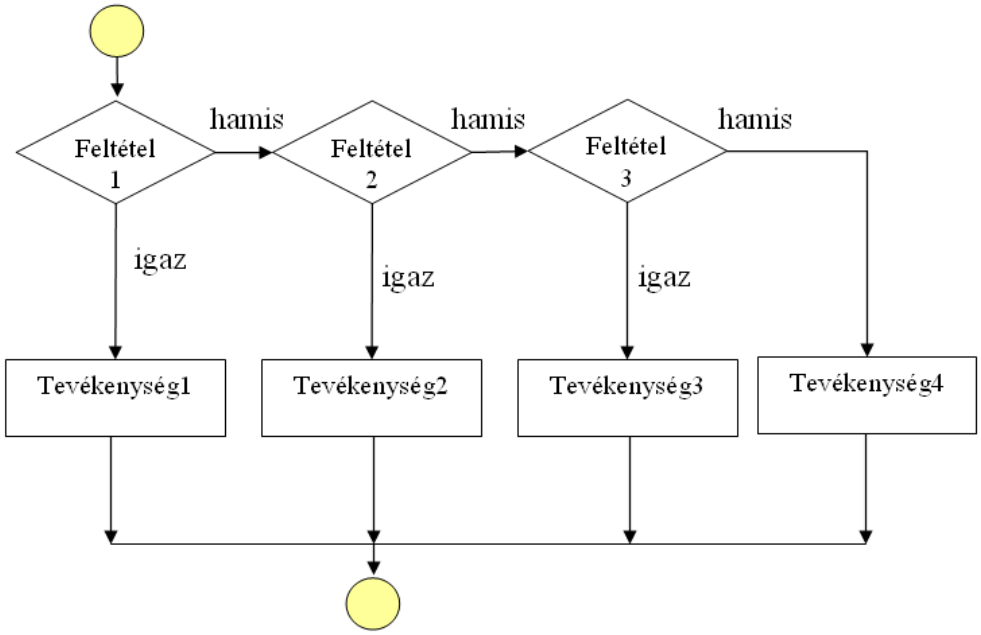
Többágú elágazással	Kétágú elágazásokkal

Ha *feltétel1* akkor
  Tevékenység1
különben ha *feltétel2* akkor
  Tevékenység2
különben ha *feltétel3* akkor
  Tevékenység3
különben
  Tevékenység4
elágazás vége
	ha *feltétel1* akkor
	  Tevékenység1
	különben
	  ha *feltétel2* akkor
	  Tevékenység2
	  különben
	  ha *feltétel3* akkor
	  Tevékenység3
	  különben
	  Tevékenység4
	  elágazás vége
	  elágazás vége
	elágazás vége

A többágú jelölésmód szemmel láthatóan tömörebb, jobban áttekinthető leírást tesz lehetővé. A kétágú elágazásokkal megvalósított megoldás példa arra, hogy egy elágazás tevékenysége bonyolultabb tevékenységszerkezet is lehet – jelen esetben pl. a külső elágazás „különben” ága maga is egy elágazás, amelynek „különben” ága szintén egy elágazás.

2. A tabulátor (vagy szóközök) következetes alkalmazása ugyan nem kötelező, de ennek hiánya egy közepes nagyságú programot is már olvashatatlanná tesz. A megfelelően tabulált és jól olvasható kóddal elsősorban magunknak teszünk jót, de ha kódunkat közzé is akarjuk tenni, akkor feltétlenül alkalmazni kell. Ez vonatkozik minden írott kódra, akár pszeudokódban írt algoritmusról, akár valamilyen programozási nyelven írott kódról van szó. Ez utóbbi esetben az intelligensebb kódszerkesztő programok általában segítséget is adnak (automatikus kódformázás). Azonban bármely egyszerű editorban írt kódnak is megfelelően formázottnak kell lennie: a formázás alapvetően a programozó feladata. Általános irányelv: egy formázatlan program olyan fokú igénytelenségről tanúskodik, mintha egy étteremben csorba, rúzsos szélű pohárban hoznák ki az 1982-es Château la Mission Haut-Brion-t.

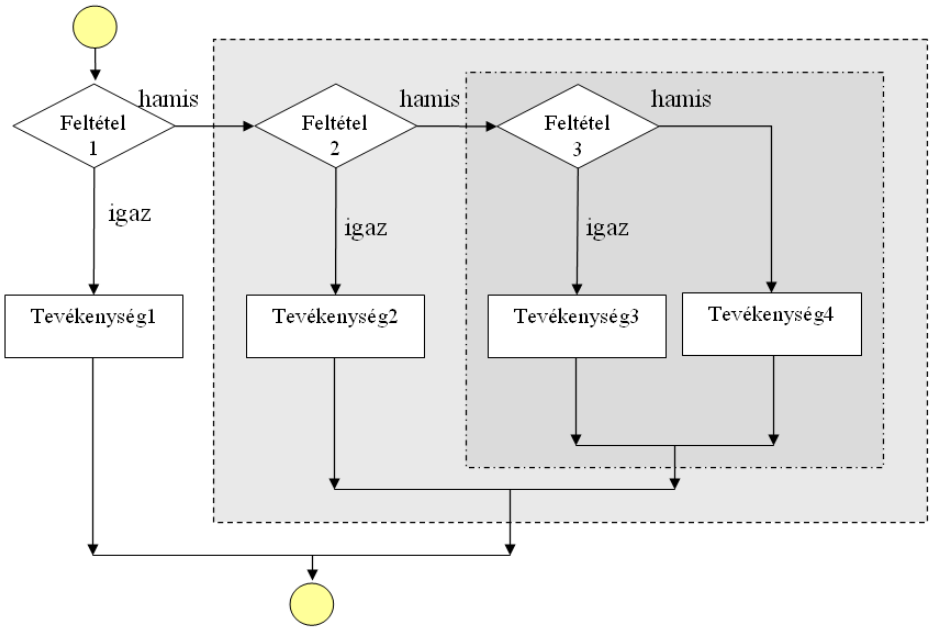
A többágú elágazások folyamatábrás jelölése a 3.4. ábrán látható. Általánosságban *K* ág megvalósításához *K* –1 elágazás-szimbólum használata szükséges.



3.4. ábra. Többágú elágazás megvalósítása folyamatábrával

A többágú elágazás folyamatábrás megvalósításához az egyetlen létező eszközt, a folyamatábra elágazás-szimbólumát használtuk, nincs speciális jelölés erre az esetre. Így nem meglepő, hogy a létrejött struktúra pontosan megegyezik az 1. megjegyzésben kétágú elágazásokból létrehozott struktúrával: valójában itt is egymásba ágyazott, egyszerű kétágú elágazásokkal van dolgunk, ami azonnal nyilvánvalóvá is válik, ha egy kicsit átrajzoljuk a 3.4. ábrát. A 3.5. ábrán a szemléletesebb jelölés kedvéért bekereteztünk egyes elemeket: láthatunk egy nagyobb, szaggatott vonallal határolt téglalapot és egy kisebb, pont-vonallal határolt téglalapot is. Ha először a nagyobb téglalap belsejét gondolatban kitöröljük (és esetleg a téglalapot kicsit kissebre is zsugorítjuk), akkor az ábra pontosan azt a képet tükrözi, ami a 3.3. ábrán látható: egy kétágú elágazást látunk, amelynek feltétele a Feltétel1, az igaz ágon a Tevékenység1 nevű te-

vékenységet hajtjuk végre, míg a hamis ágon azt a tevékenységet, ami a szaggatott vonallal jelölt téglalapban van (emlékezzünk rá: megengedtük, hogy az elágazás tevékenységei összetett tevékenységek is lehessenek). Egy nagyon fontos momentum: a szaggatott vonallal jelölt téglalapba egy nyíl vezet be és abból egy nyíl vezet tovább. Ha most figyelmünket a szaggatott vonallal jelölt téglalap belsejére fordítjuk – amelynek egyetlen belépési pontja és egyetlen kilépési pontja van – akkor itt most hasonló módon a kisebbik, pont-vonalas téglalapot gondolatban kitörölve ismét az ismerős ábrát látjuk: a szaggatott vonallal jelzett téglalap belseje nem más, mint egy kétágú elágazás, feltétele a Feltétel2, igaz ágán a Tevékenység2, hamis ágán pedig a pont-vonallal jelzett (összetett) tevékenységet hajtjuk végre. Ismét fontos: a pont-vonallal jelzett téglalapba egyetlen nyíl vezet be és egyetlen nyíl vezet ki, ezért ezt kezelhetjük egyetlen összetett tevékenységnek is. Az ábrát tovább boncolgatva észrevesszük, hogy a pont-vonallal jelzett téglalapban ismét egy kétágú elágazás van elrejtve. Vagyis a négyágú elágazást megvalósító folyamatábránk nem más, mint három darab, egymásba ágyazott kétágú elágazás. A külső (Feltétel1 feltételű) elágazás hamis ágában van egy újabb elágazás (ennek feltétele Feltétel2), amelynek hamis ága tartalmazza a harmadik elágazást (Feltétel3 feltétellel). A struktúra szó szerint tükrözi az 1. megjegyzésben leírt pszeudo-kód szerkezetét.



3.5. ábra. Többágú elágazás. A beágyazott összetett tevékenységeket téglalapok határolják.

3.3. Ciklusok

A ciklusok szolgálnak programjainkban az ismétlődő tevékenységek leírására. Az egyszerűbb ismétlődő tevékenységeket leírhatjuk ciklusok nélkül is, pl. szekvencia formájában (a kiáltsd ötször, hogy „hahó” algoritmus helyett pl. írhatjuk azt is, hogy kiáltsd, hogy „hahó”, kiáltsd, hogy „hahó”, kiáltsd, hogy „hahó”, kiáltsd, hogy „hahó”), de ez nem túl elegáns megoldás. Ráadásul lehetnek bonyolultabb esetek, ahol ezzel az egyszerű trükkel nem boldogulunk (kopogtass, amíg ajtót nem nyitnak).

A mindennapi életben is gyakran alkalmazunk ismételt tevékenységeket:

- Ússz, amíg a partot el nem éred!
- A tojásfehérjét addig keverjük, amíg kemény habot nem kapunk.
- Addig jár a korsó a kútra, amíg el nem törik.
- „Kacagj, amíg a hajnal eljön”
- Várj, amíg kinyitom az ajtót!
- Foglaljon helyet, amíg az ügyintéző felszabadul.
- Addig igyál, amíg szomjas vagy!

A ciklusok mindig tartalmaznak egy tevékenységet, valamint egy feltételt, aminek függvényében tovább folytatjuk vagy abbahagyjuk a tevékenység ismétlését. Attól függően, hogy mikor végezzük el a feltétel kiértékelését, elöltesztelő és hátultesztelő ciklusokról beszélünk. A feltételnek is két típusa lehetséges, megkülönböztetünk bennmaradási és kilépési feltételeket. A fentiek összes lehetséges variációja összesen tehát négy különféle ciklust eredményez:

- Elöltesztelő ciklus bennmaradási feltétellel
- Elöltesztelő ciklus kilépési feltétellel
- Hátultesztelő ciklus bennmaradási feltétellel
- Hátultesztelő ciklus kilépési feltétellel

Az elöltesztelő ciklusok *először* megvizsgálják, hogy kell-e (tovább) ismételni a tevékenységet, és ha igen, akkor egyszer megismétlik, majd újra ellenőrzik a feltételt. A hátultesztelő ciklusok ezzel ellentétben egyszer mindenképpen végrehajtják a tevékenységet, majd ezek után ellenőrzik, hogy szükséges-e tovább ismételni. Ha igen, akkor ismét végrehajtják azt és újra ellenőriznek. Vizsgáljuk meg egy példán, mit is jelent ez a két megközelítés.

Morc Misi ezt a jó tanácsot adja Féllábú Ferkónak: Amíg szomjas vagy, igyál rumot!

- Ha Féllábú Ferkó elöltesztelő ciklust futtat a fejében, akkor először felteszi a kérdést: szomjas vagyok? Amennyiben a válasz igen, iszik egy pohár rumot, majd ismét megkérdezi magától: szomjas vagyok? És ezt addig folytatja, amíg egyszer azt nem találja, hogy nem szomjas: ekkor nem iszik több rumot (egyelőre), végrehajtotta a tanácsot.
- Ha Ferkó hátultesztelő ciklust futtat a fejében, akkor először iszik egy pohár rumot, majd felteszi a kérdést: szomjas vagyok? Amennyiben a válasz igen, iszik egy újabb pohárral, majd ismét megkérdezi magától: szomjas vagyok? Az ivást-kérdezést addig folytatja, amíg egyszer azt a választ nem adja, hogy nem szomjas: ekkor abbahagyja az ivást, a tanácsot ismét végrehajtotta.

A fő különbség tehát az volt, hogy a hátultesztelő ciklus esetén Ferkó mindenképpen ivott egy pohár rumot (függetlenül attól, hogy szomjas volt-e vagy sem), majd ezután tette csak fel a „szomjas vagyok-e” kérdést. Az elöltesztelő ciklus esetén már az első pohár előtt feltette a kérdést, így előfordulhatott az az eset is, hogy nem ivott egy pohárral sem, mert egyáltalán nem volt szomjas (ez persze a kalózok esetén pusztán elméleti lehetőség, ők a tapasztalatok szerint mindig szomjasak).

A bennmaradási és a kilépési feltétel közti különbséget a következő példával illusztrálhatjuk:

Morc Misi, a rettegett kalózvezér egy sikeres zsákmány után a következő szónoklattal indítja a lakomát:

Cimborák, egyetek, amíg éhesek vagytok!

A fedélzetmester a következőképpen fordítja le ezt érthetőbb nyelven a matrózoknak:

Naplopó népség, ha jóllaktatok, abbahagyni a zabálást!

Mindkét utasítás ugyanazt jelenti: amíg éhes a matróz, egyen, ha már nem éhes, ne egyen. Morc Misi megfogalmazásában egy bennmaradási feltételt használt: addig kell az evés tevékenységet folytatni (vagyis a ciklusban maradni), amíg az éhség állapota fennáll. A fedélzetmester ezzel ellentétben azt fogalmazta meg, mikor kell abbahagyni a ciklust (az evés folyamatát): akkor, ha a jóllakás állapota bekövetkezik. Ezt kilépési feltételnek nevezzük.

Ha megvizsgáljuk a bennmaradási és a kilépési feltétel közti összefüggést, akkor azt találjuk, hogy ezek egymás logikai negáltjai: éhes vagyok = NEM vagyok jól lakva, vagy: jól vagyok lakva = NEM vagyok éhes. A bennmaradási és a kilépési feltételek tehát igen könnyen átalakíthatók egymásba.

Az elöltesztelő ciklusokat pszeudo-kóddal a következő módokon lehet leírni:

Elöltesztelő ciklus, bennmaradási feltétellel:

```
ciklus amíg feltétel1
      tevékenység
ciklus vége
```

A pszeudo-kódos leírásban az elöltesztelő ciklust a `ciklus` kulcsszó nyitja és a `ciklus vége` kulcsszó zárja. A bennmaradási feltétel jelzésére az `amíg` kulcsszót alkalmazzuk. A tevékenység itt is természetesen lehet összetett tevékenység is (pl. egy szekvencia, elágazás, egy másik ciklus, vagy ezek kombinációi). Az ismételt tevékenységet a *ciklus magjának* nevezzük. Ebben a leírásban a ciklusmagot mindaddig végrehajtjuk, amíg *feltétel1* igaz. A feltételt a mindig a ciklusmag végrehajtása előtt értékeljük ki. Amint *feltétel1* hamissá válik, a ciklusból kilépünk.

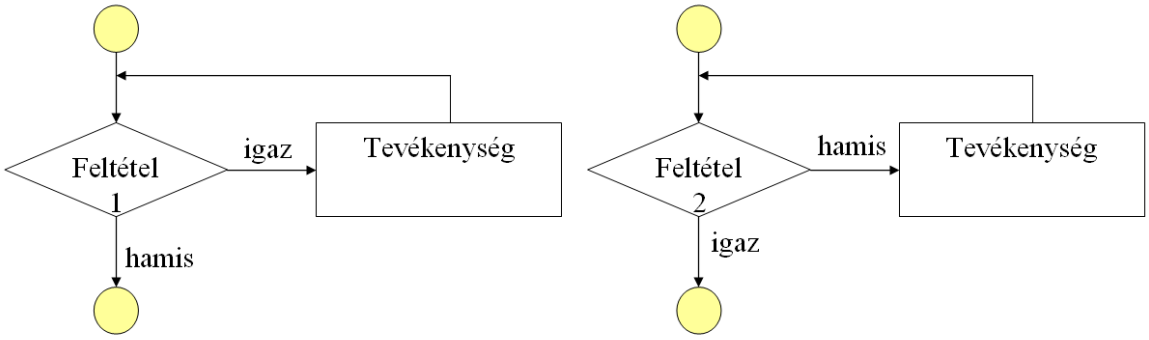
Elöltesztelő ciklus, kilépési feltétellel:

```
ciklus mígnem feltétel2
      tevékenység
ciklus vége
```

Itt az `amíg` kulcsszó helyett a kilépési feltétel jelzésére a `mígnem` kulcsszót alkalmaztunk. Ebben a ciklusban a ciklusmag mindaddig újra és újra végrehajtódik, amíg a *feltétel2* hamis (a feltétel kiértékelése itt is a ciklusmag végrehajtása előtt történik). Amint feltétel2 igazzá válik, a ciklusból kilépünk. Természetesen itt is igaz, hogy *feltétel1* = NEM *feltétel2*.

Az elöltesztelő ciklusok folyamatábrás reprezentációit a 3.6. ábra mutatja. Jól látható, hogy a bennmaradási feltétel esetében a feltétel igaz ága mutat a ciklusmag felé, míg a kilépési feltétel esetében a feltétel igaz ága a ciklusból való kilépésre mutat. A folyamatábra jól mutatja, hogy a végrehajtás során először történik meg a feltétel kiértékelése, majd ezután következhet a ciklusmag végrehajtása vagy a ciklusból való kilépés. A folyamatábrából az is világosan látszik, hogy van olyan lehetséges végrehajtási út, ami elkerüli a ciklusmagot:

az elöltesztelő ciklus akár 0-szor is lefuthat (abban az esetben, ha már a ciklus indulásakor hamis a bennmaradási, vagy igaz a kilépési feltétel).



3.6. ábra. Elöltesztelő ciklus bennmaradási (Feltétel1) és kilépési (Feltétel2) feltétellel

A hátultesztelő ciklusokat pszeudo-kódos leírásai a következők:

Hátultesztelő ciklus, bennmaradási feltétellel:

```
ciklus
      tevékenység
amíg feltétel1
```

A hátultesztelő ciklust szintén a `ciklus` kulcsszó nyitja de a ciklus zárása az és az `amíg` kulcsszóval és a hozzá tartozó bennmaradási feltétel megadásával történik (itt tehát nincs szükség a `ciklus vége` kulcsszóra). A ciklusmagot mindaddig végrehajtjuk, amíg *feltétel1* igaz, de itt a feltételt a ciklusmag végrehajtása után értékeljük ki. Amint *feltétel1* hamissá válik, a ciklusból kilépünk.

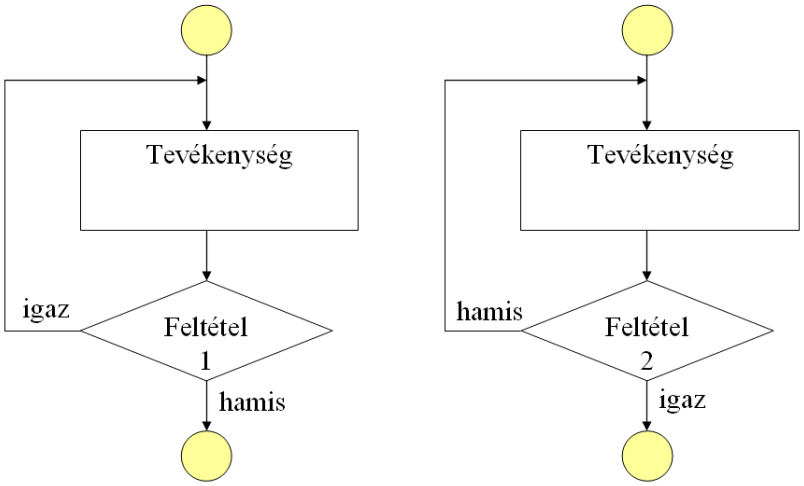
Hátultesztelő ciklus, kilépési feltétellel:

```
ciklus
      tevékenység
mígnem feltétel2
```

A ciklus zárása itt is a feltétel megadásával történik, de itt – mivel kilépési feltételt használunk – a `mígnem` kulcsszót használjuk. A ciklusban a ciklusmagot mindaddig újra és újra végrehajtjuk, amíg a *feltétel2* hamis. A feltétel kiértékelése itt is a ciklusmag végrehajtása után történik meg. Amint feltétel2 igazzá válik, a ciklusból kilépünk. Természetesen itt is igaz a bennmaradási és kilépési feltételek közötti *feltétel1* = NEM *feltétel2* összefüggés.

Az hátultesztelő ciklusok folyamatábrás leírásai a 3.7. ábrán láthatók. Itt a ciklus végrehajtása a ciklusmaggal kezdődik, majd a feltétel kiértékelésével folytatódik. A bennmaradási és kilépési feltételek itt is egymás negáltjai: a bennmaradási feltétel igaz ága marad a ciklusban, míg a kilépési feltétel igaz ága lép ki a ciklusból. A jelölésből jól látszik, hogy a ciklusmagot mindig legalább egyszer végrehajtjuk, hiszen az első vizsgálat csak a ciklusmag első végrehajtása után történik meg, a ciklust elhagyására itt nyílik először lehetőség.

Fontos ismételten hangsúlyozni, hogy hasonlóan a szekvencia és elágazás tevékenység-szerkezetekhez, a ciklusnak is egyetlen belépési pontja és egyetlen kilépési pontja van. Ezen belépési és kilépési pontokat a folyamatábrán kis körök jelzik.



3.7. ábra. Hátultesztelő ciklus bennmaradási (Feltétel1) és kilépési (Feltétel2) feltétellel

Felmerülhet a kérdés: miért van szükség ennyiféle ciklusra? Nem használhatnánk a négyféle ciklus helyett csak egyet, esetleg kettőt? Ha igen, akkor melyik ciklust, vagy ciklusokat választuk ki a négy közül egy minimalista programozási nyelvhez? Először vizsgáljuk meg a bennmaradási-kilépési feltételek kérdését. Azt már megállapítottuk, hogy ezek mindig egymás negáltjai. Tehát ha programozási nyelvünk ezek közül pl. csak a bennmaradási feltételt alkalmazza, akkor azt olyan algoritmusokat, amikben kilépési feltételt alkalmazó ciklus van, könnyűszerrel átalakíthatjuk bennmaradási feltételes ciklussá úgy, hogy közben a feltételt negáljuk. És ez fordítva is igaz. Így valójában elég vagy egyik, vagy másik feltétel, és az is mindegy, hogy melyiket alkalmazzuk.

Vajon az elöl és hátultesztelő ciklusok közül elég-e csak az egyik? Ha igen, melyiket kell alkalmazni? A fő különbség az, hogy hol történik a feltétel vizsgálata. Ennek egyik jól látható hatása az volt, hogy a hátultesztelő ciklusnál mindig végrehajtódik legalább egyszer a ciklusmag, míg az elöltesztelő ciklusnál előfordulhat, hogy egyszer sem hajtjuk azt végre. Ha csak elöltesztelő ciklusunk van, akkor egy hátultesztelő ciklust könnyűszerrel szimulálhatunk így (pl. bennmaradási feltétellel):

```
tevékenység
ciklus amíg feltétel1
    tevékenység
ciklus vége
```

A ciklusmag megismétlésével az elöltesztelő ciklus előtt pontosan ugyanazt a hatást értük el, mint a hátultesztelő ciklus esetén. Próbáljuk meg a fordított esetet: helyettesítsük az elöltesztelő ciklust egy hátultesztelő segítségével. Az alábbi megoldás egy elágazás segítségével oldja meg a problémát:

```
ha feltétel1 akkor
    ciklus
        tevékenység
    amíg feltétel1
elágazás vége
```

Tehát az elöl- vagy hátultesztelő ciklusokból is elég lenne az egyik (a példák tanúsága szerint talán egyszerűbb az élet egy elöltesztelő ciklussal). Vagyis elegendő egyetlen egy ciklust megvalósítani a programozási nyelvben és ennek segítségével kis ügyeskedéssel az összes lehetséges ciklusfajtát már meg tudjuk valósítani. Ennek ellenére a legtöbb programozási nyelv tipikusan nyújt egy elöltesztelő és egy hátultesztelő ciklust is, hogy a programozó a feladatához leginkább illeszkedő fajtát választhassa. Az alábbi táblázat példákat ad a négy lehetséges ciklusra különféle programozási nyelvekből.

	Elöltesztelő	*Hátultesztelő*

Bennmaradási feltétel	**Java:**
	`while(`*`feltétel`*`){`
	`  ciklusmag`
	`}`
		C:
		`do{`
		`  ciklusmag`
		`}while(`*`feltétel`*`)`

Kilépési feltétel	**Scratch:**
	`Repeat until` *`feltétel`*
	`  ciklusmag`
		Pascal:
		`Repeat`
		`  ciklusmag`
		`until` *`feltétel`*

A legtöbb programozási nyelv tartalmaz olyan ciklusokat, amelyeket előre meghatározott számúszor lehet lefuttatni (általában ezeket számlálóvezérelt, vagy for-ciklusoknak hívják). Ezek a ciklusok valójában nem egy újabb típust alkotnak, hanem csak egyszerűsített jelölést kínálnak egy gyakran alkalmazott problémára. Pszeudo-kódban így jelölhetjük a fix számú ciklusokat:

```
ciklus számláló=0-tól (N-1)-ig
     tevékenység
ciklus vége
```

Ebben a ciklusban a ciklusmag pontosan N-szer hajtódik végre, ahol N egy nem-negatív egész szám, közben pedig a ciklusszámláló egyesével növekszik. Ezt a ciklusfajtát a programozási nyelvek egy elöltesztelő ciklussal valósítják meg. Gyakorlásként valósítsuk meg mi is ezt a ciklust egy elöltesztelő ciklussal, bennmaradási feltétel alkalmazásával:

```
számláló=0
ciklus amíg számláló < N
     tevékenység
     számláló = számláló + 1
ciklus vége
```

3.4. Strukturált program

Strukturált programnak nevezzük az olyan programot, amely csak a három alapvető tevékenységszerkezetet használja. Ez nagyon erős megkötésnek tűnik, hiszen mi van akkor, ha egy probléma megoldására nem lehet ilyen módon algoritmust készíteni? Szerencsére ez alaptalan aggodalom, mert bizonyítható, hogy minden algoritmikusan megoldható probléma megoldható strukturált programmal is (Böhm-Jacopini tétel). Ezért nincs okunk arra, hogy ne strukturáltan programozzunk. Természetesen előfordul, hogy egy feladat egyszerűbben megfogalmazható, ha alkalmazunk más eszközöket is (pl. ugrás típusú utasításokat), de ezeket az eszközöket csak azoknak ajánljuk, akik már jól és magabiztosan tudnak programozni, valamint pontosan fel tudják mérni, hogy strukturáltság feláldozása megéri-e az esetleges egyéb előnyök tükrében.

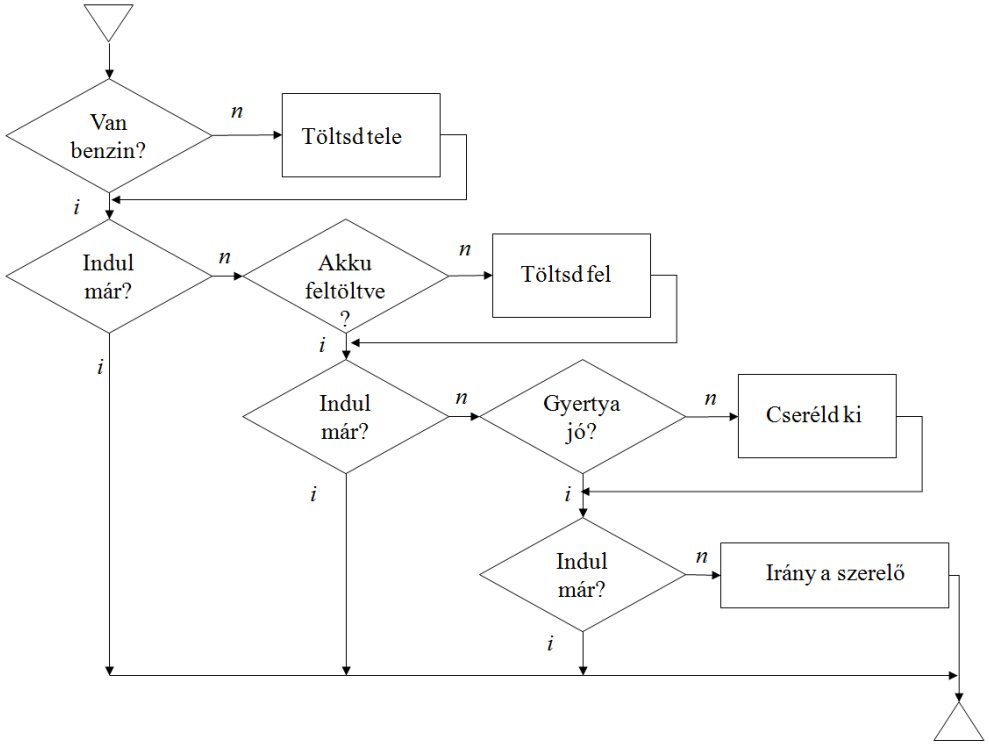
Készítsük el most az 2.2. ábrán látható másodfokú egyenlet-megoldó programot pszeudokódos leírással:

```
eljárás másodfokú egyenlet
      beolvas: a
      beolvas: b
      beolvas: c
      legyen D = b^2 - 4ac
      ha D < 0 akkor
          kiír „Nincs valós gyök”
      különben
          legyen x1 = (-b + √D) / 2a
          legyen x2 = (-b - √D) / 2a
          kiír: „az első gyök” x1
          kiír: „a második gyök” x2
      elágazás vége
eljárás vége
```

A fenti program kizárólag szekvencia és elágazás tevékenységszerkezeteket használ, tehát nyilvánvalóan kielégíti a strukturált program definícióját. Figyeljük meg, az egyes tevékenységszerkezetekbe beágyazott tevékenyszerkezeteket: a program utasításszekvenciájának egyik eleme egy elágazás, amely maga is tartalmaz (a különben ágban) egy szekvenciát.

A 2.3. ábrán az ALG2.3 algoritmus folyamatábrás reprezentációja látható. Vizsgáljuk meg ezt az aránylag jól áttekinthető, világos algoritmust és próbáljuk azonosítani a három tevékenységszerkezetet. Sajnos próbálkozásunk nem fog sikerrel járni: a 2.3. ábra nem strukturált program. A folyamatábra rajzolása közben nem figyeltünk arra, hogy az elágazásokat az elágazás tevékenységszerkezet szabályainak megfelelően rajzoljuk fel. Hiába próbálkozunk az átrajzolással, nem fogjuk tudni azonosítani az elágazások tevékenységeit (pl. a 3.5. ábra szerinti módon). Ez az algoritmus nem strukturált programmal lett megvalósítva. Készítsük el

most már szigorúan a három tevékenységszerkezet alkalmazásával az ALG2.3 algoritmust. Az eredeti, kicsit pongyola megfogalmazású algoritmus kicsit más értelmezéséből adódó lehetséges megoldás a 3.8. ábrán látható. Itt már jól láthatóan minden elágazás egy elágazás tevékenységszerkezet is: egyetlen belépési ponttal és egyetlen kilépési ponttal.



3.8. ábra. A motorkerékpár-ápolás kocáknak (ALG2.3) strukturált megvalósítása– folyamatábrával

A folyamatábrával megadott algoritmust felírhatjuk pszeudo-kóddal is. Az alábbi pszeudo-kód pontosan a 3.8. folyamatábra logikáját tükrözi:

```
eljárás kocamotoros_strukturált
      ha nincs benzin akkor
          töltsd tele
      ha nem indul akkor
          ha akku nincs feltöltve akkor
              töltsd fel
          elágazás vége
          ha nem indul akkor
             ha gyertya nem jó akkor
                 cseréld ki
             elágazás vége
             ha nem indul akkor
                 irány a szerelő
             elágazás vége
          elágazás vége
      elágazás vége
eljárás vége
```

Figyeljük meg, hogy a folyamatábra önmagában lehetőséget adott olyan algoritmusok készítésére is, amelyek nem csak a három alapvető tevékenységszerkezetet tartalmazzák. Ha figyelünk a szabályokra, akkor természetesen a folyamatábra segítségével is tudunk strukturált programot készíteni, de a folyamatábra nem kényszerít rá erre. Ha a három tevékenységszerkezetet tartalmazó pszeudo-kóddal fogalmazzuk meg algoritmusunkat, akkor bizonyosan strukturált programot fogunk kapni, itt nem tudunk kilépni a szabályok közül: a pszeudo-kód rákényszerít a strukturált programozásra. Az a szabadság azonban, amit a folyamatábra egyes építőelemei közötti nyilak tetszőleges húzogatása jelent, egészen kusza programok megírását teszi lehetővé. Ez az elem, ami az építőelemek tetszőleges sorrendbeli végrehajtását teszi lehetővé, sok nyelvben valamilyen ugró (pl. goto) utasításként van jelen. A goto használata – ha van ilyen az általunk használt programozási nyelvben – egy strukturált programban természetesen tilos.

Az ALG2.1 algoritmus x és y számok ($x \geq y > 0$) legnagyobb közös osztójának meghatározására szolgál. Az algoritmust a következőképpen fogalmaztuk meg:

Legyen r az x és y maradékos osztásának maradéka.

Az x helyébe tegyük az y-t, y helyére az r-t.

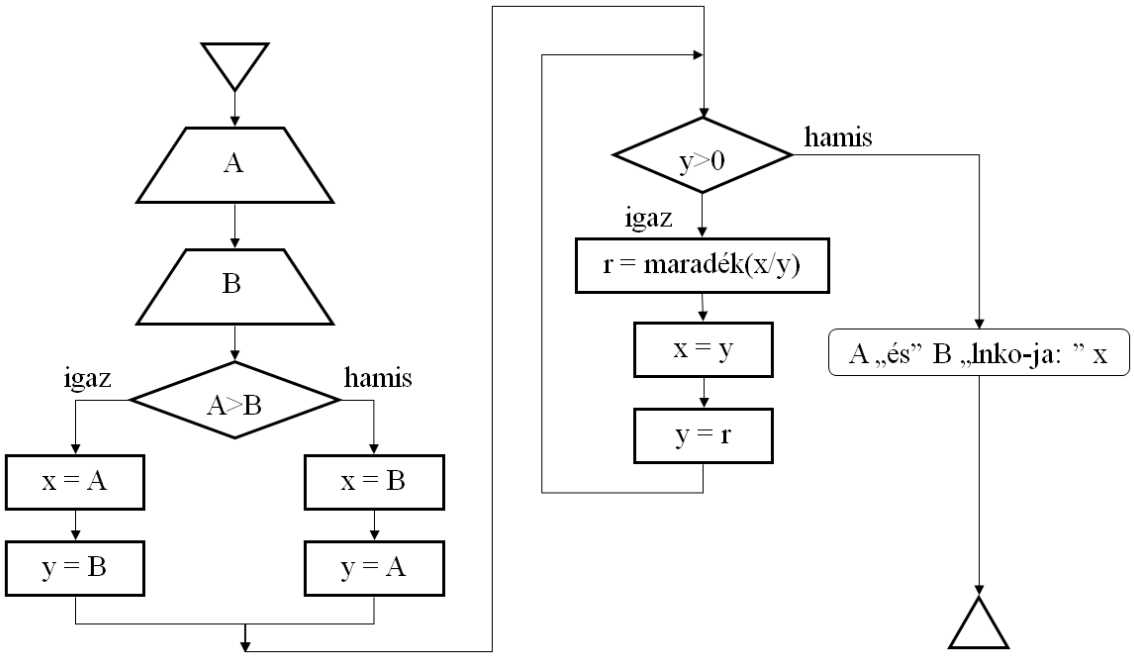
Ha $y = 0$, akkor x a legnagyobb közös osztó, különben ismételjük az 1. lépéstől.

Készítsük most el ezen algoritmus leírását strukturált programmal, pszeudo-kóddal és folyamatábrával is. Ahhoz, hogy a program ne csak $x \geq y$ esetben működjön, a beolvasás után szükség szerint cseréljük meg a két számot.

Az algoritmus először beolvas két számot (*A*-t és *B*-t), majd ezeket az x és y változókba tölti oly módon, hogy immár $x \geq y$ igaz legyen. Ezután kezdődik a legnagyobb közös osztó számítása egy ciklus segítségével: a ciklus addig fut, amíg y nullára nem csökken, ezen idő alatt minden iterációban a ciklus magjában elvégezzük a maradékos osztást, valamint az x és y változók áttöltését. Végül a program kiírja az eredményt.

```
  eljárás Euklideszi algoritmus
1     beolvas: A
2     beolvas: B
      ha A≥B akkor
3           legyen x értéke A
4           legyen y értéke B
      különben
5           legyen x értéke B
6           legyen y értéke A
      elágazás vége
      ciklus amíg y>0
7           legyen r x/y maradéka
8           legyen x értéke y
9           legyen y értéke r
      ciklus vége
      kiír A „és” B „lnko-ja:” x
  eljárás vége
```

Az algoritmus folyamatábrás megvalósítása a 3.9. ábrán látható. A folyamatábra pontosan a pszeudo-kódban leírt lépéseket követi: a program egy szekvencia, melynek elemei a két beolvasás, az elágazás, a ciklus, majd a kiírás. Az elágazás mindkét ágában egy-egy kételemű szekvencia található, míg a ciklus magja egy háromelemű szekvencia.

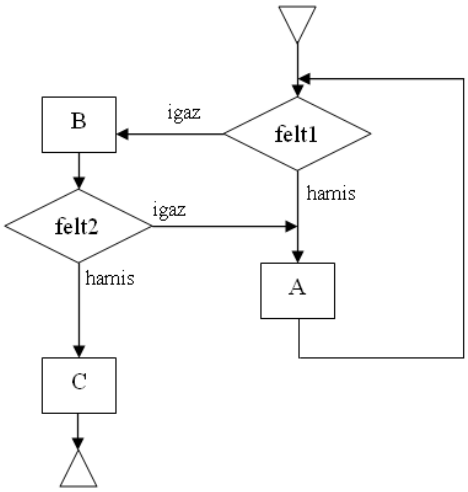


3.9. ábra. Az euklideszi algoritmus folyamatábrája

A program működését az alábbi példán követhetjük nyomon. A program végrehajtása során a 84 és 30 számokat adjuk meg bemenetként, amelyek legnagyobb közös osztója a 6, amit a program végrehajtása után meg is kapunk. Az alábbi táblázat a program egyes változóinak értékeit mutatja a program végrehajtása során. Az első oszlopban álló „lépés” index azonos a pszeudo-kódos leírásban a sorok előtt álló számokkal.

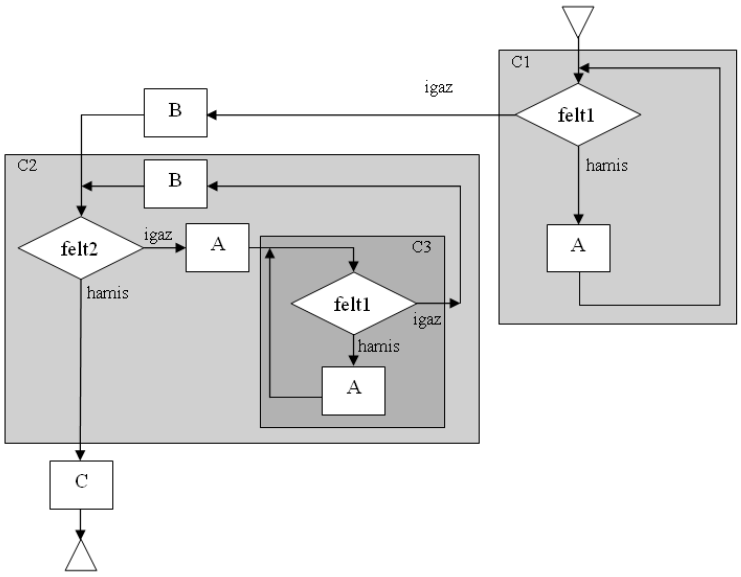
Lépés	*A*	*B*	*x*	*y*	*r*
1	84				
2	84	30			
3	84	30	84		
4	84	30	84	30	
7	84	30	84	30	24
8	84	30	30	30	24
9	84	30	30	24	24
7	84	30	30	24	6
8	84	30	24	24	6
9	84	30	24	6	6
7	84	30	24	6	0
8	84	30	6	6	0
9	84	30	**6**	0	0

A 3.10. ábra egy nehezen áttekinthető folyamatábrát tartalmaz, melyben két elágazás-szimbólum és három egyéb tevékenység (A, B és C) található. A folyamatábra struktúrái emlékeztetnek a ciklusokra, de ezek nem szabályosak, nem tudjuk elkülöníteni a szabványos tevékenységszerkezeteket.



3.10. ábra. Egy kesze-kusza folyamatábra

Némi átalakítás után a 3.11. ábra szerinti folyamatábrát kaphatjuk, amely pontosan ugyanazt csinálja, mint a 3.10. ábrán látható, de ebben már jól felismerhetők a tevékenységszerkezetek. A könnyebb láthatóság érdekében a folyamatábra három ciklusát (C1, C2, C3) téglalapokkal határoltuk. A C1 ciklus egy elöltesztelő ciklus, melynek magja az A tevékenység. A C2 ciklus szintén elöltesztelő, ennek magja egy szekvencia, melynek elemei az A tevékenység, a C3 ciklus és a B tevékenység. A C3 ciklus is elöltesztelő, magja pedig az A tevékenység. Maga a program egy szekvencia, melynek elemei a C1 ciklus, a B tevékenység, a C2 ciklus és a C tevékenység. A ciklusok közül a C1 és C3 kilépési feltétellel, míg a C2 ciklus bennmaradási feltétellel rendelkezik.

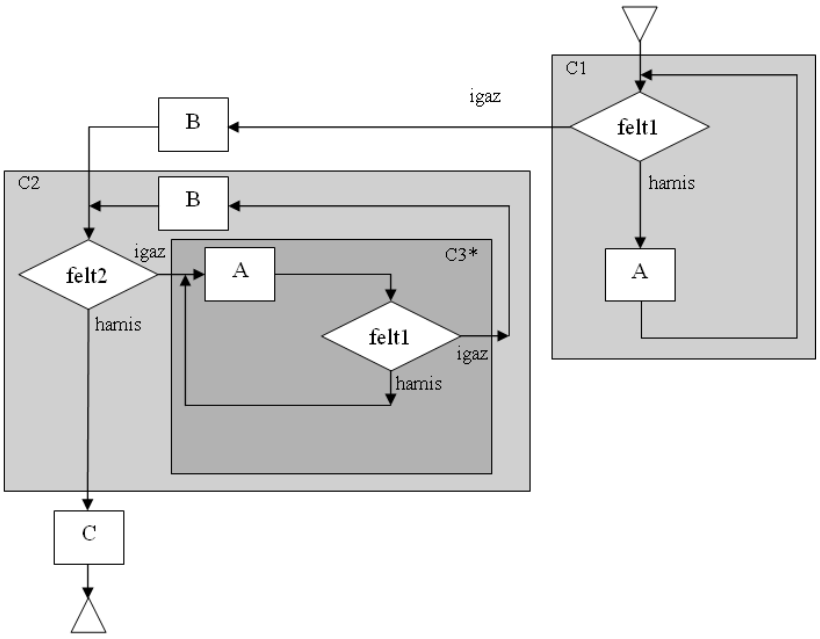


3.11. ábra. A 3.10. folyamatábra átalakítva, a ciklusok (C1, C2, C3) külön jelölve

A 3.11. ábrán látható programot pszeudo-kóddal is leírhatjuk a következő módon:

```
eljárás Átalakított_program_1
    ciklus mígnem felt1
        végrehajt A
    ciklus vége
    végrehajt B
    ciklus amíg felt2
        végrehajt A
        ciklus mígnem felt1
            végrehajt A
        ciklus vége
        végrehajt B
    ciklus vége
    végrehajt C
eljárás vége
```

Ha jól megfigyeljük a C3 ciklust, láthatjuk, hogy annak ciklusmagja (az A tevékenység) megismétlődik a ciklus előtt is. Ez egy további egyszerűsítést tesz lehetővé, ahogy a 3.11. ábrán látható: az A tevékenységből és a C3 elöltesztelő ciklusból álló szekvenciát a C3* hátultesztelő ciklussal helyettesíthetjük.



3.12. ábra. A 3.11. folyamatábra egyszerűsítve, a C3 ciklus és környezete a C3 ciklussá alakítva*

Az egyszerűsített program pszeudo-kódja a következő:

```
eljárás Átalakított_program_2
    ciklus mígnem felt1
        végrehajt A
    ciklus vége
    végrehajt B
    ciklus amíg felt2
        ciklus
            végrehajt A
        mígnem felt1
        végrehajt B
    ciklus vége
    végrehajt C
eljárás vége
```

Feladatok:

3.1. Tegyük fel, hogy mostantól csak a legegyszerűbb elágazást használhatjuk: `ha` *`feltétel`* `akkor – elágazás vége` formában, a `különben` kulcsszó pedig nem létezik. Írjuk meg a következő algoritmust ebben a környezetben, pszeudo-kódban: ha szép idő van kirándulok, különben olvasgatok.

Valósítsuk meg a 3.4. ábrán látható négyágú elágazást is ezzel a módszerrel. Írjuk fel az algoritmust pszeudo-kóddal, illetve rajzoljuk fel folyamatábrával is.

3.2. Gyakorlásképpen rajzoljuk át a 3.8. ábrán látható folyamatábrát úgy, hogy bejelöljük rajta (pl. bekeretezzük) az egyes elágazásokhoz tartozó összetett tevékenységeket (amelyeknek természetesen egy belépési pontja és egy kilépési pontja lesz). A szép ábra érdekében a 3.8. ábra alsó részét kicsit át kell rajzolni.

3.3. Írjuk át az euklideszi algoritmust úgy, hogy hátultesztelő ciklust használjon. Egészítsük ki továbbá ellenőrzéssel is: csak pozitív egész számokat fogadjon el bemenetül.

3.4. Az euklideszi algoritmus egy alternatív megfogalmazása a következő:
Legyen r az x és y maradékos osztásának maradéka. Ha $r = 0$, akkor y a legnagyobb közös osztó, különben x helyébe tegyük az y-t, y helyére az r-t és ismételjük az első lépéstől.
Valósítsuk meg ezt az algoritmust is pszeudo-kóddal és folyamatábrával is (strukturáltan).

3.5. Alakítsuk át a 3.11. ábrán látható algoritmus pszeudo-kódját úgy, hogy csak elöltesztelő ciklust használjunk bennmaradási feltétellel.
Végezzük el az átalakítást úgy is, hogy csak kilépési feltételt használunk.

3.6. Alakítsuk át a 3.12. ábrán látható algoritmus pszeudo-kódját úgy, hogy csak elöltesztelő ciklust használjunk bennmaradási feltétellel.
Végezzük el az átalakítást úgy is, hogy csak kilépési feltételt használunk.

3.7. Írjuk fel az alábbi idézeteket formálisan pszeudo-kód és folyamatábra segítségével.

- „Ha a világ rigó lenne, / kötényemben ő fütyülne, / éjjel-nappal szépen szólna, / ha a világ rigó volna.”

- „Ha a napnak lába volna, / bizonyára gyalogolna.”
- „Tudod, Jancsi szivem, örömest kimennék, / Ha a mosással oly igen nem sietnék”
- „E világon ha ütsz tanyát, / hétszer szűljön meg az anyád!”
- „Ha a Pál kéménye füstöl, / Péter attól mindjár’ tüszköl; / Ellenben a Péter tyukja / Ha kapargál / A szegény Pál / Háza falát majd kirugja”
- „Kacagj, amíg a hajnal eljön”
- „Bujdosva jártam a világot széltére, / Mígnem katonának csaptam föl végtére.”
- „Belekapaszkodott, el sem szalasztotta, / S nagy erőlködéssel addig függött rajta, / Mígnem a felhő a tengerparthoz ére, / Itten rálépett egy szikla tetejére.”
- „Csak addig várjon, amíg / Egy szócskát, egy kicsi szócskát mondok – / Aztán verjen kelmed agyon, / Ha jónak látja.”
- „Az amazontermészetü Márta / Hős elszántsággal törte magát át / A vívó tömegen, / Mígnem férjére talált”
- „S kezeit könyörögve kinyujtá / Az amazontermészetü Mártához, / De ez nem könyörült. / Megfogta egyik lábszárát, / S kihúzta az asztal alól, / És addig döngette, amíg csak / A seprőnyélben tartott. / Aztán ott ragadá meg haját, / Ahol legjobban fáj, / És elhurcolta magával, / Mint a zsidó a lóbőrt”
- „Azért én élni fogok, / Míg a világnak / Szappanbuboréka / Szét nem pattan.”
- „Míg a mező dermed, fázik, / a zöld fenyves csak pompázik.”
- „Rab vagy, amíg a szíved lázad”

4. fejezet

Konverzió pszeudo-kódok és folyamatábrák között

Mint láttuk, a folyamatábra lehetőséget ad nem strukturált programok készítésére, míg a pszeudo-kód általunk definiált elemkészlete szigorúan csak a három alapvető tevékenységszerkezet használatát teszi lehetővé, így garantálja, hogy csak strukturált programot írjunk. Ebből azonnal következik, hogy nem létezhet kölcsönösen egyértelmű megfeleltetés a folyamatábrával és pszeudo-kóddal leírható algoritmusok között. Azonban ha a folyamatábra készítésénél betartjuk a szabályt, hogy az elágazás-szimbólumot csak szabályos tevékenységszerkezetek (elágazás vagy ciklus) készítésére használjuk, akkor ezen folyamatábra már kölcsönösen egyértelműen hozzárendelhető egy pszeudo-kódhoz.

A pszeudo-kód átírása folyamatábrává egyszerű feladat, hiszen a pszeudo-kódban egyértelműen jelölve vannak a tevékenységszerkezetek: ha azt olvassuk, hogy „ha”, akkor tudjuk, hogy ez elágazás, míg a „ciklus” szó egy iterációt jelöl, sőt a szintaxisból azonnal látjuk, hogy elöl- vagy hátultesztelő ciklusról van-e szó, és kilépési vagy bennmaradási feltételt használ-e. Ezen tevékenységszerkezeteknek létezik egyértelmű megfelelője a folyamatábrában is, az átrajzolás többé-kevésbé mechanikus feladat. A folyamatábra átírása pszeudo-kóddá nehezebb feladat, hiszen a folyamatábra elágazás-szimbóluma többféle tevékenységszerkezetet jelölhet; először azonosítani kell az elágazás-szimbólum szerepét (elágazást vagy ciklust valósít-e meg, cikluson beül pontosan melyik fajtát), majd ezután a pszeudo-kód generálása már ismét egyszerű tevékenység.

4.1. Pszeudo-kód átalakítása folyamatábrává

A pszeudo-kódban szereplő egyes elemi tevékenységeket általában a folyamatábra téglalap szimbólumával jelöljük, ez alól az adatbevitel és adatkiírás kivételek: ezek jelölésére használjuk a folyamatábra speciális szimbólumait. Az összetett tevékenységszerkezeteket (ciklus, elágazás) át kell alakítani: a folyamatábrában ezeket az elágazás-szimbólumok és egyéb elemi tevékenység segítségével ábrázoljuk.

Szekvenciák átalakítása. A szekvencia átalakítása egyszerű: a szekvencia egyes tevékenységeit – ha azok elemi tevékenységek – nyilak segítségével kössük össze. Ha a szekvencia tevékenységei között vannak összetett tevékenységek is, akkor az összetett tevékenységet először helyettesítsük az azt jelölő összetett tevékenységgel (téglalappal) és így húzzuk be az összetett

tevékenységbe befutó és onnan kifutó nyilat. Ezután az összetett tevékenységet ki kell fejteni (lásd alább). A program mindig egy szekvencia (elfajuló esetben csak egyetlen tevékenységből áll, és általában ennek a szekvenciának a tevékenységei összetett tevékenységek); ne feledkezzünk el a folyamatábra elejére elhelyezni a start-, végére pedig a stop szimbólumot.

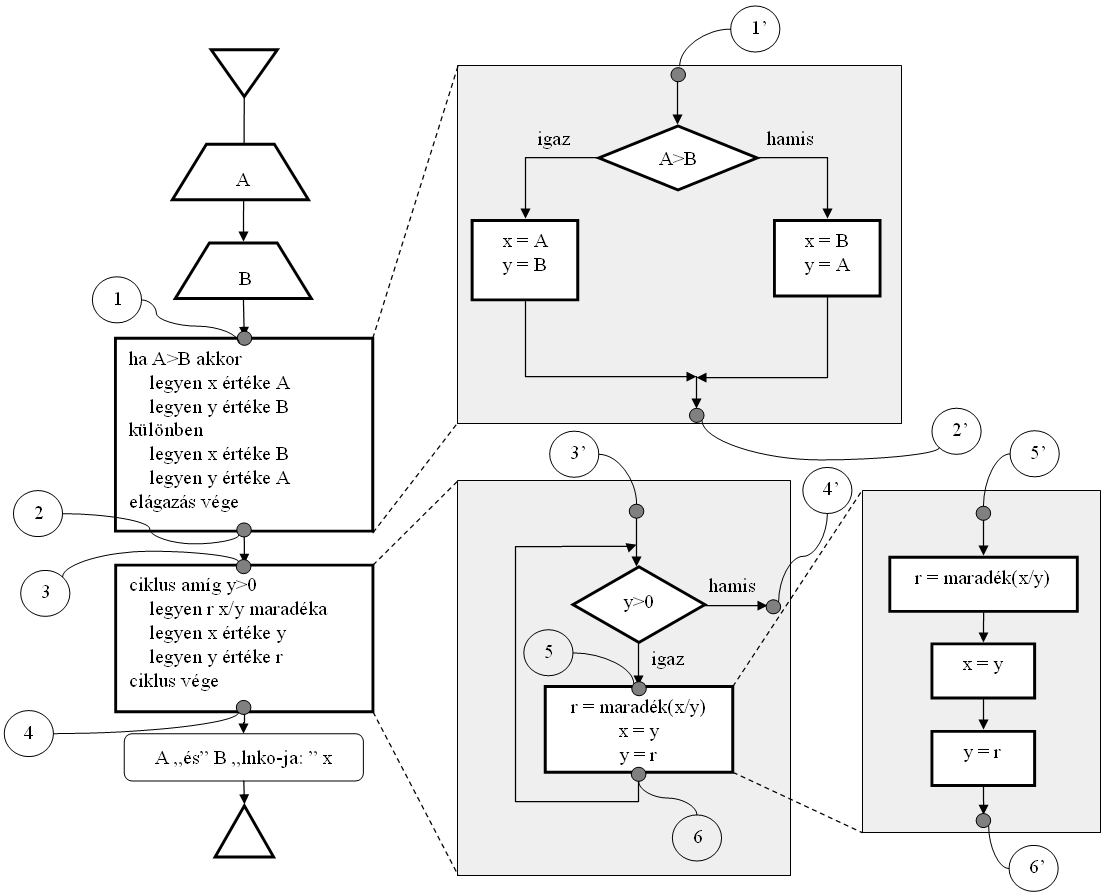
Elágazások átalakítása. Az elágazások átalakítása során először határozzuk meg, hány ágú elágazásról van szó és válasszuk ki a megfelelő struktúrát (pl. a 3.2.–3.4. ábrák szerint). A pszeudo-kódban levő feltételeket írjuk be a folyamatábra elágazás-szimbólumaiba és írjuk fel az elágazásokra az igaz-hamis értékeket is. Ezután az elágazások egyes ágainak átalakítása történik. Minden ágat kezdetben helyettesítsünk egyetlen összetett tevékenységgel (téglalappal), majd ezeket szükség szerint fejtsük ki (a kifejtést lásd alább).

Ciklusok átalakítása. A pszeudo-kód egyértelműen azonosítja, hogy elöl- vagy hátultesztelő ciklusról, illetve kilépési vagy bennmaradási feltételről van-e szó. Ezek alapján válasszuk ki a megfelelő struktúrát a ciklus folyamatábrás ábrázolására (a 3.6. és 3.7. ábrák szerint). Írjuk be az elágazás szimbólumba a (kilépési vagy bennmaradási) feltételeket és az igaz-hamis értékeket az elágazás szimbólumra (ügyeljünk arra, hogy kilépési vagy bennmaradási feltételt használunk-e). A ciklus magját kezdetben helyettesítsük egyetlen összetett tevékenységként (téglalappal). Ezután a ciklus magját, ha szükséges, fejtsük ki (lásd alább).

Összetett tevékenységek kifejtése. Az összetett tevékenység lehet szekvencia, elágazás, vagy ciklus, illetve ezekben további összetett tevékenységek lehetnek. Az összetett tevékenység kifejtése során először döntsük el, milyen típusú tevékenységszerkezetről van szó (szekvencia, elágazás, ciklus), majd a fentebb leírtak szerint alakítsuk át az összetett tevékenységet. Itt előfordulhat, hogy ismételten egy összetett tevékenységre bukkanunk, ezeket is fejtsük ki, amíg csupa elemi tevékenységet tartalmaz a folyamatábra. Ezen kifejtett összetett tevékenység bizonyosan egyetlen belépési és egyetlen kilépési ponttal rendelkezik, azonosítsuk ezeket. A (még ki nem fejtett) összetett tevékenységbe mutató nyilat a kifejtett összetett tevékenység belépési pontjára, illetve a (még ki nem fejtett) összetett tevékenységből kifelé mutató nyilat a kifejtett összetett tevékenység kilépési pontjára kell átrajzolni.

4.1. Példa: alakítsuk át az Euklideszi algoritmus pszeudo-kódos leírását folyamatábrává:

```
eljárás Euklideszi algoritmus
      beolvas: A
      beolvas: B
      ha A≥B akkor
          legyen x értéke A
          legyen y értéke B
      különben
          legyen x értéke B
          legyen y értéke A
      elágazás vége
      ciklus amíg y>0
          legyen r x/y maradéka
          legyen x értéke y
          legyen y értéke r
      ciklus vége
      kiír A „és” B „lnko-ja:” x
eljárás vége
```



4.1. ábra. Az euklideszi algoritmus átalakításának lépései.

A program egy szekvencia, melynek elemei: két egymást követő beolvasás, egy elágazás, egy ciklus és egy kiírás. Ezt a szekvenciát, az összetett tevékenységszerkezeteket egyelőre megőrizve (és a start-stop szimbólumokat hozzáadva) átrajzolhatjuk a 4.1. ábrának megfelelően. Az átalakítás első lépése után rendelkezésünkre áll egy szekvencia (a start és stop jelek között), amelynek két eleme olyan összetett tevékenység, amely további kifejtésre vár. Az első ilyen összetett tevékenység egy kétágú elágazás, amelynek mindkét ágában egy-egy összetett tevékenység (kételemű szekvencia) áll. Ennek az elágazásnak, mint összetett tevékenységnek a belépési pontját 1-el jelöltük az ábrán, míg a kilépési pontját 2 jelöli. A kifejtett elágazás a felső szürke téglalapban található, ez szerkezetileg a 3.3. ábrán látható elágazásnak felel meg, de már tartalmazza a feltételt és az elágazás egyes ágait is: ezek jelen esetben két-két értékadást jelentenek. Az elágazás egyes ágait tovább is lehetne bontani, de a példában itt megállunk. A kifejtett elágazás belépési pontját 1’, míg kilépési pontját 2’ jelöli, ezt kell behelyettesíteni az eredeti ábrába úgy, hogy az 1 pont az 1’ ponthoz, míg a 2 pont a 2’ ponthoz illeszkedjen.

A 4.1. ábra kiinduló szekvenciájában található egy másik kifejtendő tevékenységszerkezet: egy elöltesztelő ciklus, bennmaradási feltétellel. Ennek megfelelő szerkezetet a 3.6. ábra bal oldalán láthatunk: ezt felhasználva és behelyettesítve a feltételt és a ciklusmagot kapjuk a 4.1. ábra bal alsó szürke téglalapjában látható kifejtett ciklus. Ennek belépési pontja a 3’, kilépési pontja a 4’, ezeket nyilvánvalóan a kiinduló ábra 3 és 4 pontjainak kell megfeleltetni.

A ciklus magja egy összetett tevékenység (egy szekvencia), ez tovább bontható, ahogy a jobb alsó szürke téglalapban látszik. Ezt a kifejtett sorozatot az 5 és 6 pontok közé kell illeszteni úgy, hogy az 5' és 6' pontok legyenek az illeszkedő be- és kilépési pontok. A kifejtett összetett tevékenységek behelyettesítése után a 3.9. ábrával megegyező eredményre jutunk.

4.2. Folyamatábra átalakítása pszeudo-kóddá

Láttuk, hogy a folyamatábrával nem strukturált programok is könnyen rajzolhatók, amelyek a Böhm-Jacopini tétel szerint mindig átalakíthatók strukturált kóddá. Most nem ezen átalakítás a célunk (erre a 3.4. fejezetben láttunk példát), hanem feltételezzük, hogy a folyamatábra már eleve csak strukturált tevékenységszerkezeteket tartalmaz: ilyen strukturált folyamatábrákat alakítunk át pszeudo-kóddá.

A folyamatábra – pszeudo-kód átalakítás azért nehéz feladat, mert a folyamatábra elágazás-szimbóluma akár elágazás tevékenységszerkezetet, akár ciklust jelölhet. Az átalakítás során az első és legfontosabb tevékenységünk annak azonosítása lesz, hogy milyen szerepet játszanak az ábrában szereplő egyes elágazás-szimbólumok. Amennyiben a fő tevékenységszerkezeteket sikerült azonosítani, ezek pszeudo-kóddá való átírása már mechanikus feladat.

4.2.1. A folyamatábra átírása pszeudo-kóddá

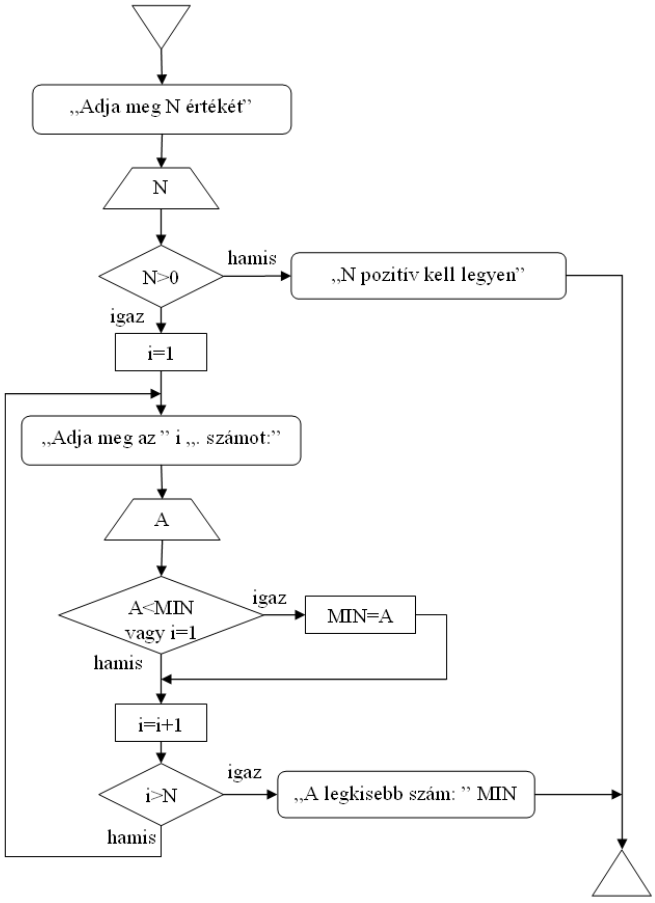
Az átalakítást az elágazások és ciklusok azonosításával kezdjük. A folyamatábra minden elágazás-szimbóluma egy összetett tevékenységszerkezet részét képezi, ami lehet elágazás vagy ciklus. Minden egyes ilyen szimbólumról egyenként el kell dönteni, hogy milyen szerepet játszik (lásd az *Elágazások azonosítása*, *Elöltesztelő ciklusok azonosítása* és *Hátultesztelő ciklusok azonosítása* pontokat). A tevékenységszerkezetek azonosítása után következik a pszeudo-kód generálása. A program maga egy szekvencia, amely a start és a stop szimbólumok között foglal helyet (elfajuló esetben egyelemű szekvencia is lehet, de általában több, ráadásul összetett tevékenységből áll). A start szimbólum helyett írjuk fel az `Eljárás eljárás_név` kulcsszó-azonosító párost, a stop szimbólum helyett pedig az `Eljárás vége` kulcsszót. A jól olvashatóság érdekében ezen két kulcsszót a papír bal szélére (de mindenképpen pontosan egymás alá) kell írni, a megfelelő térközt kihagyva a program számára. Ezután a start és stop szimbólum közötti szekvencia átírása következik (a *Szekvenciák átírása* pont szerint), kicsit beljebb kezdve (pl. tabulátort vagy néhány szóközt használva).

Elágazások azonosítása. Az elágazások sémái a 3.2., 3.3. és 3.4. ábrákon láthatók; ezen sémákra illeszkedő folyamatábra-részletek biztosan elágazás tevékenységszerkezeteket jelölnek. Mivel a több, mint kétágú elágazás a folyamatábrában egymásba ágyazott (legfeljebb) kétágú elágazásokból áll (lásd a 3.5. ábrát), elegendő az egyszerű, egyetlen elágazás-szimbólumot tartalmazó tevékenységszerkezeteket azonosítani. Ha szükséges, később az egymásba ágyazott elágazásokra az egyszerűsített pszeudo-kódos jelölés alkalmazható. Az elágazások közös jellemzője, hogy a tevékenységszerkezet belépési pontja egy elágazás-szimbólum, majd a vezérlés két ágon folytatódik (ezek közül az egyik lehet üres tevékenység is, mint a 3.2. ábrán, vagy mindkét ágon végezhetünk tevékenységet, mint a 3.3 ábra mutatja), majd e két végrehajtási út egyetlen ponton találkozik, ami egyben a tevékenységszerkezet kilé-

pési pontja is lesz. Az elágazás-szimbólum tehát akkor és csak akkor jelöl elágazás tevékenységszerkezetet, ha az elágazás-szimbólum mindkét kilépési pontja után tudunk azonosítani egy-egy olyan összetett tevékenységszerkezetet, amelyek vezérlése egyetlen pontban találkozik. Ezen összetett tevékenységszerkezetek azonosítása történhet úgy, hogy gondolatban bekeretezzük a tevékenységszerkezetet, amelynek egyetlen belépési pontja van (amely az elágazás-szimbólumhoz csatlakozik) és egyetlen kilépési pontja pedig a másik ág kilépési pontjához csatlakozik. Fontos, hogy ezen azonosított összetett tevékenységeket határoló (gondolatban) felrajzolt téglalapot a belépési pont és a kilépési pont kivételével nem lépheti át más nyíl.

Elöltesztelő ciklusok azonosítása. Az elöltesztelő ciklusok sémái a 3.6. ábrán láthatók. Az elöltesztelő ciklus belépési pontja agy elágazás-szimbólum bemenete, kilépési pontja pedig ugyanezen elágazás-szimbólum egyik kimenete. Az elágazás-szimbólum másik kimenete után egy (összetett) tevékenység következik (ciklusmag), melynek kilépési pontja a ciklus belépési pontjához csatlakozik. Egy elágazás-szimbólum tehát akkor és csak akkor jelöl elöltesztelő ciklust, tudunk azonosítani egy olyan összetett tevékenységszerkezetet (ciklusmagot), amelynek belépési pontja az elágazás-szimbólum egyik kimenetéhez, kilépési pontja pedig az elágazás-szimbólum bemenetéhez csatlakozik. Ekkor az elöltesztelő ciklus tevékenységszerkezet belépési pontja az elágazás-szimbólum bemenete, kilépési pontja pedig az elágazás-szimbólum másik (nem a ciklusmaghoz csatlakozó) kimenete lesz. Amennyiben a ciklusmagba az elágazás-szimbólum igaz ága vezet, bennmaradási, ellenkező esetben kilépési feltételről van szó (lásd a 3.6. ábrát).

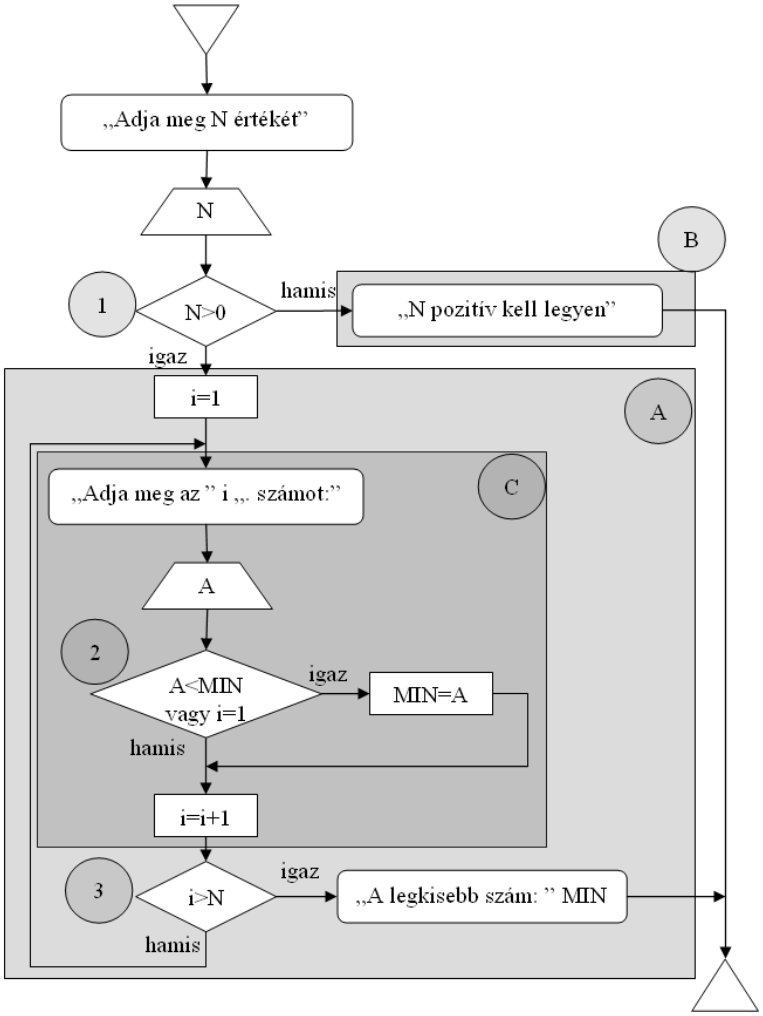
Hátultesztelő ciklusok azonosítása. Az hátultesztelő ciklusok sémái a 3.7. ábrán láthatók.



4.2. ábra. A minimumkereső program folyamatábrája

A hátultesztelő ciklus belépési pontja mindig egy (összetett) tevékenység, amelynek kilépési pontja egy elágazás-szimbólum bemenetéhez csatlakozik. Ezen elágazás-szimbólum egyik kimenete a ciklus belépési pontjához csatlakozik, a másik kimenete pedig a hátultesztelő ciklus tevékenységszerkezet kilépési pontja lesz. Egy elágazás-szimbólum tehát akkor és csak akkor jelöl hátultesztelő ciklust, tudunk azonosítani egy olyan összetett tevékenységszerkezetet (ciklusmagot), amelynek kilépési pontja az elágazás-szimbólum bemenetéhez, belépési pontja pedig az elágazás-szimbólum egyik kimenetéhez csatlakozik. Ekkor a hátultesztelő ciklus tevékenységszerkezet belépési pontja a ciklusmag belépési pontja lesz, a tevékenységszerkezet kilépési pontja pedig az elágazás-szimbólum másik (a ciklusmaghoz nem csatlakozó) kimenete lesz. Amennyiben a ciklusmaghoz az elágazás-szimbólum igaz ága csatlakozik, akkor bennmaradási, ellenkező esetben pedig kilépési feltételt használunk (lásd a 3.7. ábrát).

Szekvenciák átírása. A szekvencia elemei lehetnek elemi tevékenységek (ide értve a beolvasást, a kiírást, vagy a folyamatábra téglalap szimbólumaiba írt egyéb tevékenységeket), vagy lehetnek ciklusok és elágazások. A szekvencia átírása során egyszerűen az egyes tevé-



4.3. ábra. A minimumkereső programban feltárt tevékenységszerkezetek

kenységeket egymás alá írjuk. Amennyiben a szekvenciában elágazás vagy ciklus is szerepel résztevékenységként, azokat az *Elágazások átírása* és a *Ciklusok átírása* pontok szerint fejtjük ki. Ügyeljünk arra, hogy a szekvencia írása során az egyes tevékenysége pontosan egymás

alá kerüljenek: ez elemi tevékenységek esetén triviális, elágazások és ciklusok esetén pedig az összetett tevékenységszerkezet nyitó és záró kulcsszava legyen a többi tevékenységszerkezethez igazítva.

Elágazások átírása. Az elágazásként azonosított tevékenységszerkezet átírását a `ha` *`feltétel`* `akkor` kulcsszó – feltétel – kulcsszó hármassal kezdjük és az `elágazás vége` kulcsszó leírásával kezdjük. Ha szükséges (valódi kétágú elágazás), akkor használjuk a `különben` kulcsszót is. A kulcsszavakat pontosan egymás alá írjuk. A feltétel helyére írjuk be az elágazás-szimbólumban található feltételt. Ezután az egyes végrehajtási ágak kifejtése történik a *Szekvenciák átírása* pont szerint. Az egyes végrehajtási ágak kifejtését kezdjük beljebb. Ügyeljünk arra az egyes végrehajtási ágak helyes behelyettesítésére: a folyamatábrán lévő *igaz* ág az `akkor` kulcsszó után , a *hamis* ág pedig a `különben` kulcsszó után következik.

Ciklusok átírása. Miután pontosan azonosítottuk a ciklus és a feltétel típusát, írjuk le a ciklust nyitó és záró kulcsszavakat a megfelelő feltétellel együtt:

A feltétel megegyezik a folyamatábra elágazás-szimbólumának feltételével. A ciklus vázának felírása után a ciklusmag kifejtése történik a *Szekvenciák átírása* pontban leírtak szerint. A ciklus magjának leírását kezdjük beljebb.

`ciklus amíg feltétel`	(elöltesztelő ciklus, bennmaradási feltétellel)
`ciklus vége`	
`ciklus mígnem feltétel`	(elöltesztelő ciklus, kilépési feltétellel)
`ciklus vége`	
`ciklus`	(hátultesztelő ciklus, bennmaradási feltétellel)
`amíg` *`feltétel`*	
`ciklus`	(hátultesztelő ciklus, kilépési feltétellel)
`mígnem` *`feltétel`*	

4.2. Példa: Alakítsuk át a 4.2. ábrán látható minimumkereső program folyamatábráját pszeudo-kóddá.

Első lépésként azonosítsuk a folyamatábra elágazás-szimbólumaihoz rendelhető tevékenységszerkezeteket. A 4.3. ábrán számokkal jelölt elágazás-szimbólumok szerepe a következő: Az 1. jelű elágazás-szimbólum egy elágazást valósít meg, ahol az igaz ág az A jelű, míg a hamis ág a B jelű szürke téglalapba keretezve látható. A 2. jelű elágazás-szimbólum egy egyszerű elágazás, melynek igaz ága a MIN = A utasítás, a hamis ága pedig üres. A 3- jelű elágazás-szimbólum egy hátultesztelő ciklust valósít meg, aminek magja a C jelű bekeretezett területen látható. A hátultesztelő ciklus kilépési feltételt használ (hiszen a hamis ág vezet vissza ciklusmag elé).

Az elágazások és ciklusok feltérképezése után következhet a program átírása pszeudo-kóddá. A program maga egy szekvencia, amelynek elemei egy kiírás, egy beolvasás, valamint egy elágazás. Írjuk fel a pszeudo-kódot ezekkel a tevékenységekkel. Az szekvencia egyetlen összetett tevékenysége, az elágazás, csak vázként kerül be a kódba:

```
eljárás Minimumkereső (1. lépés)
      kiír: „Adja meg N értékét”
      beolvas: N
      ha N > 0 akkor
          ELÁGAZÁS KIFEJTENDŐ
      különben
          ELÁGAZÁS KIFEJTENDŐ
      elágazás vége
eljárás vége
```

Ezután következhet a vázként feljegyzett elágazás egyes ágainak kifejtése. Az igaz ág (A jelű téglalap a 4.3. ábrán) egy szekvencia, melynek elemei egy értékadás, egy ciklus, valamint egy kiírás. A hamis ágon egyetlen kiírás (B jelű téglalapban) található. A kifejtett elágazással bővített pszeudo-kód a következő:

```
eljárás Minimumkereső (2. lépés)
      kiír: „Adja meg N értékét”
      beolvas: N
      ha N > 0 akkor
          i:=1
          ciklus
            CIKLUSMAG KIFEJTENDŐ
          mígnem i > N
          kiír: „A legkisebb szám: ” MIN
      különben
          kiír: „N pozitív kell legyen”
      elágazás vége
eljárás vége
```

Ebben a leírásban egy ciklus váza található, de a ciklus magja még hiányzik. A ciklusmag ismét egy szekvencia, melynek elemei egy kiírás, egy beolvasás, egy elágazás és egy értékadás (C jelű téglalap a 4.3. ábrán). Ezzel a ciklusmaggal bővített pszeudo-kód a következő:

```
eljárás Minimumkereső (3. lépés)
      kiír: „Adja meg N értékét”
      beolvas: N
      ha N > 0 akkor
          i:=1
          ciklus
              kiír „adja meg az ” i „. számot”
              beolvas: A
              ha A < MIN vagy i = 1 akkor
                  ELÁGAZÁS KIFEJTENDŐ
              elágazás vége
              i := i + 1
          mígnem i > N
          kiír: „A legkisebb szám: ” MIN
      különben
          kiír: „N pozitív kell legyen”
      elágazás vége
eljárás vége
```

Végül az elágazás hiányzó (egyetlen) ágát kell kifejteni, ami egy egyelemű szekvencia (egy értékadó utasítás). Ezen tevékenységgel az utolsó hiányos tevékenységszerkezet is teljessé válik, a program teljes pszeudo-kódos leírása a következő:

```
eljárás Minimumkereső
      kiír: „Adja meg N értékét”
      beolvas: N
      ha N > 0 akkor
          i=1
          ciklus
              kiír „adja meg az ” i „. számot”
              beolvas: A
              ha A < MIN vagy i = 1 akkor
                  MIN = A
              elágazás vége
              i := i + 1
          mígnem i > N
          kiír: „A legkisebb szám: ” MIN
      különben
          kiír: „N pozitív kell legyen”
      elágazás vége
eljárás vége
```

Feladatok:

4.1. Alakítsuk át lépésenként az Euklideszi algoritmus 3.9. ábrán látható folyamábrás reprezentációját pszeudo-kóddá.

4.2. Alakítsuk át lépésenként a Minimumkereső program pszeudo-kódját folyamatábrává.

4.3. Alakítsuk át lépésenként a 3.11. és 3.12. ábrákon látható folyamábrákat pszeudo-kódokká.

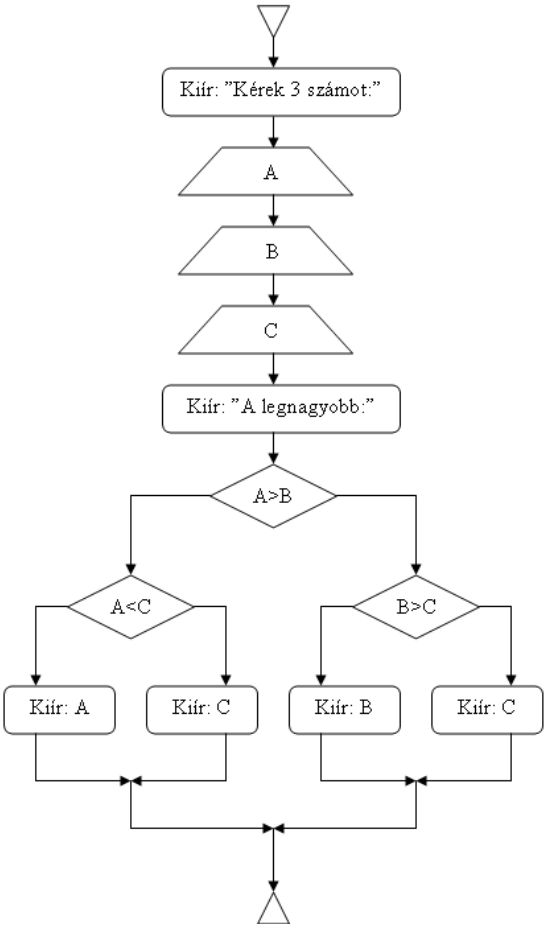
4.4. Alakítsuk át a következő pszeudo-kódot folyamatábrává:

```
Eljárás Összegző
    i := 0
    Ciklus
        S := 0
        vége := hamis
        Ciklus amíg vége = hamis
            Beolvas: k
            Ha k > 0 akkor
                S:=S+k
            Különben
                vége := igaz
            elágazás vége
        Ciklus vége
        Kiír: S
        i := i + 1
    Mígnem i = 10
Eljárás vége
```

4.5. Alakítsuk át a következő pszeudo-kódot folyamatábrává:

```
Eljárás összegzés2
    Kiír „Adja meg az összeadandó számok számát:”
    Beolvas: N
    I := 0
    D := 0
    Ciklus amíg I<N
        Beolvas: A
        D := D+A
        I := I+1
    Ciklus vége
    Kiír ”Az összeg:”
    Kiír D
Eljárás vége
```

4.6. Egészítsük ki a következő folyamatábrát az elágazások igaz-hamis értékének jelzésével, hogy a program a beolvasott három szám közül a legnagyobb értékét írja ki. Alakítsuk át a folyamatábrát pszeudo-kóddá.



5. fejezet

További eszközök tevékenység- és adatszerkezetek leírására

5.1. Adatszerkezetek és tevékenységszerkezetek

Eddig készített programjainkban arra koncentráltuk, hogy hogyan fogalmazzuk meg a szükséges tevékenységeket ahhoz, hogy a program az általunk kitűzött feladatot megoldja. Ezeket a tevékenységeket tevékenységszerkezetekbe szerveztük a strukturált programozás jegyében és többféle leírási módot is alkalmaztunk ezek jellemzésére.

A programok általában adatokon operálnak: egyszerű programjainkban ezen adatok eddig csupán néhány változót jelentettek. Bonyolultabb programok esetén az adatoknak sajátos, a probléma lényegét kifejező szerkezete lesz, ezt a szerkezetet is meg kell tervezni, illetve ennek leírására is eszközök szükségesek.

A pszeudo-kód egy szöveges forma algoritmusok, programok működésének leírására. Ez a leírás kiválóan alkalmas vezérlési szerkezetek leírására, viszont nem ad módot a programjainkban alkalmazott adatszerkezetek modellezésére. A folyamatábra szintén jól használható vezérlési szerkezetek szemléletes, grafikus leírására, azonban ennek a leírási módnak – mint azt láttuk a korábbiakban – van egy nagy hátránya: nem kényszeríti ki a strukturált megoldásokat, lehetőséget ad kesze-kusza, nem strukturált eszközöket alkalmazó programok írására. A folyamatábra ezen kívül szintén nem teszi lehetővé az adatszerkezetek leírását.

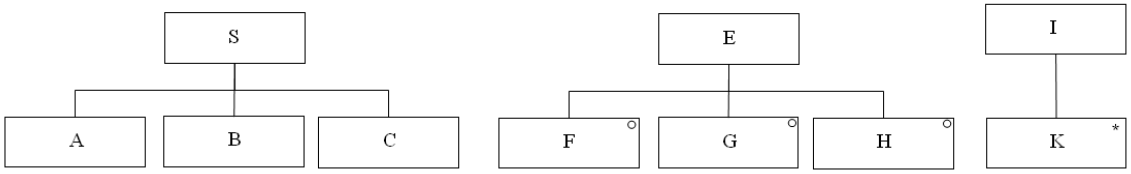
Ebben a fejezetben két olyan eszközt ismertetünk, amelyek lehetővé teszik mind a vezérlési, mind az adatszerkezetetek leírását. A Jackson-ábrák grafikus eszközkészletet nyújtanak mind a vezérlési-, mind az adatszerkezetek szemléletes leírásához. A reguláris kifejezések tömör, szöveges eszközkészletet nyújtanak, amelyek segítségével – a strukturált elemekre fókuszálva – lehet (főként) adatszerkezeteket leírni.

5.2. Jackson-ábrák

A Jackson-ábra Michael A. Jackson nevéhez fűződik, aki az 1970-es években a strukturált programtervezés kérdéseivel foglalkozott: ehhez a módszertanhoz dolgozta ki a Jackson-

ábrákat, mint grafikus segédeszközt. Mi most a Jackson-módszerrel, mint tervezési módszertannal nem foglalkozunk, viszont a Jackson-ábrák jelölésrendszerét alkalmazni fogjuk mind vezérlési szerkezetek, mind adatszerkezetek modellezésére.

A Jackson-ábrák úgynevezett operációkból állnak, amelyek jelölésére a téglalap szolgál (lásd az 5.1. ábrát.) Az operáció jelenthet egy program vezérlési szerkezetében egy egyszerű tevékenységet, vagy használhatjuk azt egy adatszerkezet egyik elemének leírására is. Ezen operációkból a három alapvető tevékenységszerkezetnek megfelelő összetett operáció állítható össze: szekvencia, szelekció és iteráció.



5.1. ábra. A Jackson-ábra elemei: szekvencia, szelekció és iteráció

Az operációk szekvenciájának jelölésére a szekvencia elemeit egymás mellé rajzoljuk, majd ezeket a szekvenciát jelző operációval összekötjük, mint azt az 5.1. ábra első példája mutatja: itt az S összetett operációt az A, B és C operációk sorozata alkotja. A Jackson-ábrán a szekvencia pontos jelentése a következő: a szekvencia minden eleme pontosan egyszer, az adott sorrendben – balról jobbra olvasva – vesz részt az összetett operációban. A példában tehát S egy A, egy B és egy C operációból áll (ebben a sorrendben).

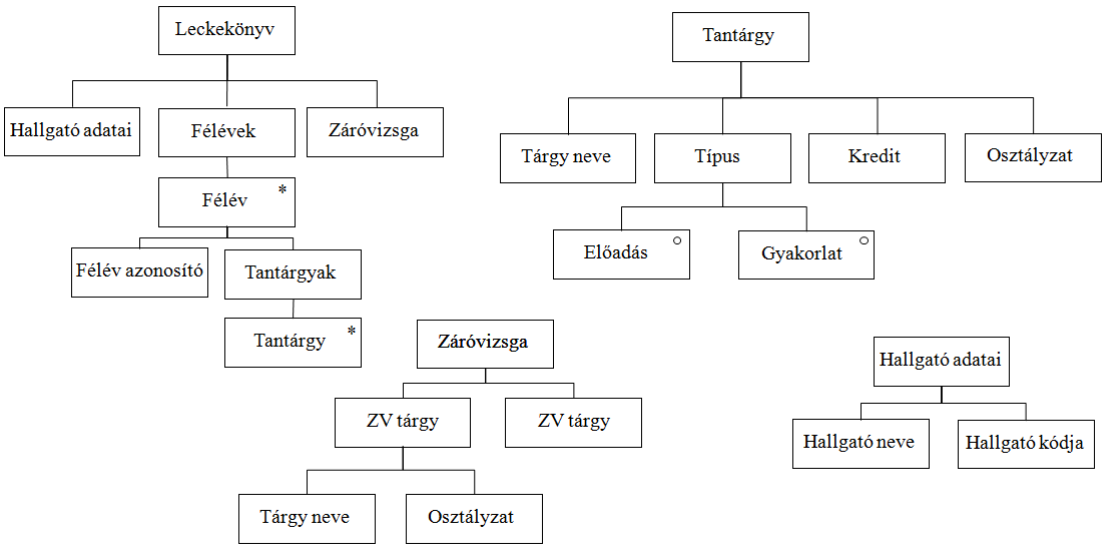
A szelekció jelölése hasonló a szekvenciáéhoz, de itt a szelekció elemeit a téglalap jobb felső sarkába rajzolt körrel jelöljük. Az 5.1. ábra második példájában az E jelű szelekció elemei az F, G és H operációk. A Jackson-ábrán szelekciójának pontos jelentése: A szelekciónak pontosan egy eleme lehet egyszerre jelen, valamilyen kiválasztási szabály szerint. A példában tehát Az E operáció vagy egy F, vagy egy G, vagy egy H operációt jelent.

Az iteráció jelölése az 5.1. ábra jobb oldalán látható: az iterációt jelölő operációt az iteráció magjával összekötjük, ahol a magot a téglalap jobb felső sarkába írt csillag jelöli. Az iteráció pontos szemantikája a Jackson-ábrán a következő: Az iteráció a mag nulla- vagy többszörös ismétléséből áll. A példában az I iteráció a K operáció ismétlése, ahol K-t 0-szor, 1-szer, 2-szer, stb. ismételhetjük.

A Jackson-ábra jelölésrendszere lehetővé teszi, hogy az egyes operációkat tovább definiáljuk. Ezt a lehetőséget vizsgáljuk meg az 5.2. ábrán látható Jackson-ábrán, ahol a leckekönyv leírása látható. Az ábra szerint a leckekönyv három komponenst tartalmaz: a hallgató adatait, a tanulmányok félévek szerinti felosztását és a záróvizsga adatait (ez egy három-elemű szekvencia). A félévek további definíciója az ábra szerint: a félévek nevű entitás több (0, 1, 2, stb.) félév nevű entitásból áll (ez egy iteráció). A félév a félév azonosítóból és a félév során tanult tantárgyakból áll (szekvencia). A tantárgyak nevű entitás jelentése: több tantárgy (szekvencia). A tantárgy nevű entitás definíciója az ábra egy másik részén található: a Jackson-ábra megengedi az diagram szétszabdalását a jobb ábrázolhatóság érdekében: az egyes entitások definíciói akárhol elhelyezhetők. Figyeljük meg: míg pl. a tantárgyak definíciója rögtön a tantárgyak nevű entitás előfordulása alatt található, addig a tantárgy nevű entitást megismételtük egy másik helyen és a definíciót ott folytattuk. A tantárgy az ábra szerint a tantárgy nevéből, típusából, a tárgy kreditszámából, valamint az elért osztályzatból áll (szekvencia).

A típust tovább definiáltuk: ez lehet előadás vagy gyakorlat, de csak az egyik (szelekció). A hallgató adatai a hallgató nevéből és kódjából állnak (szekvencia). A záróvizsgáról az index két záróvizsga tárgy adatait tárolja (szekvencia). Ezek a tárgy nevét és a kapott osztályzatot tartalmazzák (szekvencia). Figyeljük meg, hogy a záróvizsga két azonos típusú entitást tartalmaz (pontosan kettőt az ábra szerint). Ezen ZV tárgy nevű entitások azonos szerkezetűek, ezért elég, ha azt egyszer definiáljuk. A példában az egyik ZV tárgy entitást definiáltuk tovább, de megtehettük volna azt is, hogy a záróvizsga definíciója után félbehagyjuk az ábrát, majd a ZV tárgy entitást máshol megismételve folytatjuk a definíciót.

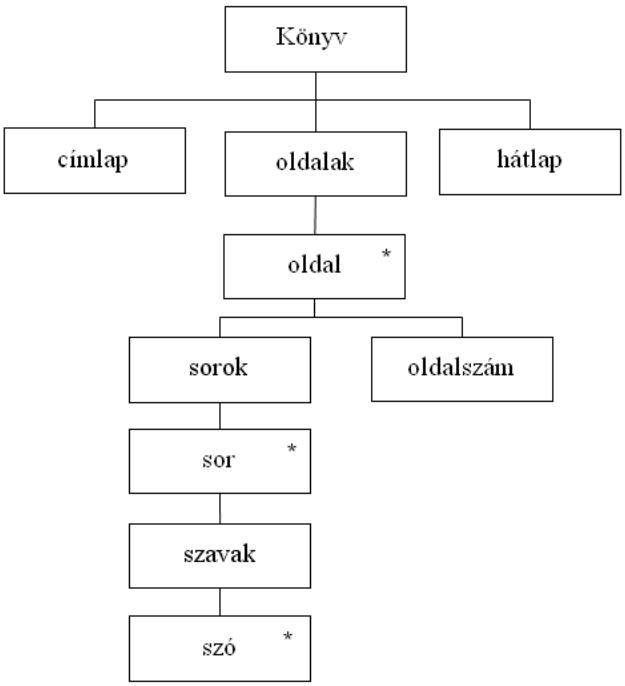
Fontos megjegyezni, hogy az egyes entitások definiálásakor mindig a felül található téglalapban *megnevezett* entitást definiáljuk: egyértelmű például, hogy az 5.2. ábrán a félévek jelzésű téglalap alatt található a félévek nevű entitás definíciója. Félreértésre adhat okot viszont a félév definíciója: a félév feliratú téglalap alatt a félév (és nem a félév*) definíciója található. Ez jól látható a tantárgy definíciójánál: a jelölés szerint egyértelmű, hogy a definíció a tantárgy entitásra vonatkozik; azonban ha alternatív módon a tantárgy definícióját annak első előfordulása alatt folytattuk volna, akkor abban a téglalapban tantárgy* felirat állna: a definíció természetesen ekkor is a tantárgy entitásra (és nem a tantárgy*-ra) vonatkozna. Hasonló megállapítás igaz a szelekció elemeinek definiálására is: ha az ábránkon pl. az előadás entitást tovább definiálnánk úgy, hogy tovább részletezzük az ábrán látható entitást, akkor a definíció értelemszerűen az előadás entitásra és nem az előadás°-ra vonatkozna.



5.2. ábra. A leckekönyv leírása Jackson-ábrával

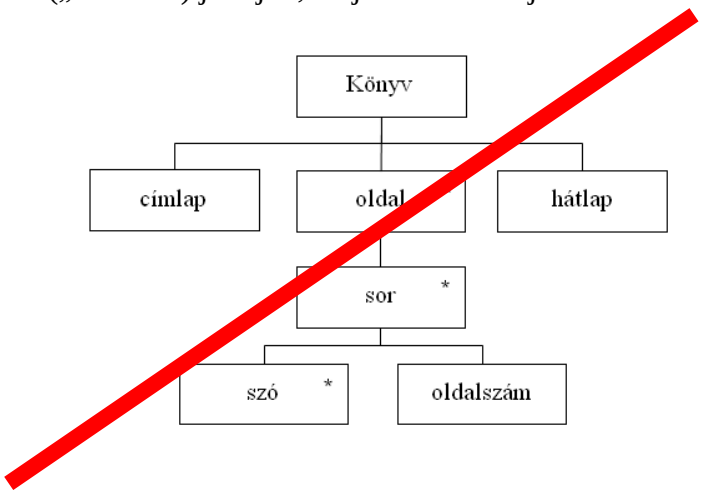
5.1. Példa: Rajzoljuk fel Jackson-ábrával az egyszerű könyv szerkezetét. Ennek a könyvnek van egy előlapja és egy hátlapja, közte pedig a könyv oldalai találhatók. Minden oldalon sorok, a sorokban pedig szavak vannak. Minden oldal alján egy oldalszám is található.

A könyv nyilván egy szekvencia formájában tartalmazza az előlapot, az oldalakat és a hátlapot (ebben a sorrendben). Az oldalak nevű entitás több oldalt tartalmaz, ez egy iteráció lesz. Az oldal nevű entitáson sorok található és egy oldalszám (szekvencia). A sorok több sorból állnak (iteráció), a sorok pedig szavakból (iteráció). A könyv Jackson-ábrája az 5.3. ábrán látható.



5.3. ábra. A könyv leírása Jackson-ábrával

A Jackson-ábrák rajzolása közben több típushibát el lehet követni: a első a szintaktikus hiba, amikor a lerajzolt ábra nem felel meg a Jackson-ábra szabályainak; a második a szemantikus hiba, amikor a lerajzolt ábra nem a modellezendő objektumot írja le. Mindkét hibára mutat példát az 5.4. ábra. Ha pusztán a rajzoló szándékát vizsgáljuk és pillanatnyilag eltekintünk a szintaktikus hibáktól, a következő jelentést fedezhetjük fel: a könyv címlapból, oldalakból és hátlapból áll, ahol az oldalak sorokból állnak, a sorok pedig szavakból (több szóból) és egy oldalszámból. Ez utóbbi definíció nagy valószínűséggel hibás: a könyvek általában nem tartalmazzák minden sor végén az oldalszámot. Ez rossz modellje a valóságnak. Vizsgáljuk most meg az alkalmazott szintaxist. A könyv entitás definíciója veti fel az első kérdést: milyen szerkezet ez? Szekvenciának nem alkalmas, mert az egyik téglalapban csillag jelzés található, iterációnak szintúgy nem jó, mert a definícióban több téglalap is szerepel, ráadásul csillag nélküliek is. Ez tehát hibás jelölés: a helyes megoldás az lenne, hogy az oldalakat (oldal*) külön szinonimaként („oldalak”) jelöljük, majd ezt definiáljuk oldal*-ként, amint azt az erede-

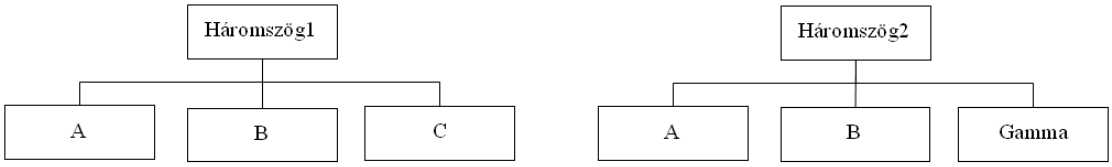


5.4. ábra. A könyv többszörösen hibás leírása Jackson-ábrával

ti megoldásban az 5.3. ábrán tettük. Hasonló jellegű hiba található a sor definíciójában is, ahol a szó* helytelenül szerepel: itt is kellene egy „szavak” nevű szinonima, amelyet aztán lehet szó*-ként tovább részletezni.

5.2. Példa: Egy háromszöget többféle adatszerkezettel is leírhatunk: megadhatjuk pl. a háromszög három oldalának hosszát (*A*, *B*, *C*), vagy megadhatjuk két oldalának hosszát (*A*, *B*) és az azok által közbezárt szöget (Gamma). Mindkét adatszerkezet egy egyszerű szekvencia, ahogy azt az 5.5. ábra mutatja. Készítsünk egy programot, amely kiszámítja a háromszög területét az első ábrázolási módot alkalmazva. A területet Héron képletével számíthatjuk ki a következőképpen:

$T = \sqrt{S(S-A)(S-B)(S-C)}$, ahol *S* a kerület fele: $S = (A+B+C)/2$.

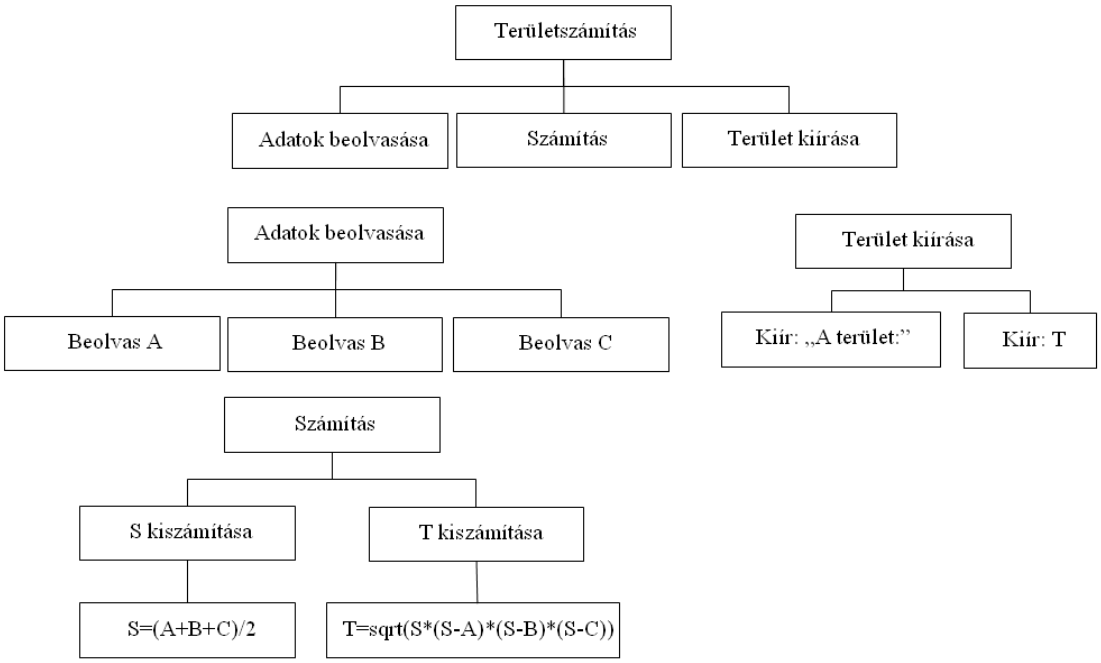


5.5. ábra. A háromszög leírására szolgáló két lehetséges adatszerkezet

Területszámító programunk a következő tevékenységeket fogja végezni:

- beolvassa a számításhoz szükséges adatokat (*A*, *B* és *C* értékét),
- elvégzi a terület kiszámítását (először *S*, majd *T* értékét számítja ki),
- kiírja a terület értékét..

A területszámító program struktúráját az 5.6. ábra mutatja.



5.6. ábra. A területszámító program leírása Jackson-ábrával

Figyeljük meg, hogy a bemenetnél használt adatszerkezet (az A, B, és C szekvencia az 5.5. ábrán) hogyan tükröződik a program beolvasó részénél: az adatszerkezet elemeinek egyértelműen megfeleltettünk a programunkban tevékenységeket. Ez gyakran történik így bonyolultabb adatszerkezetek esetén is. Azt is figyeljük meg, hogy a számítás során először az S, majd a T értékét számítottuk ki: a Jackson-ábrán a balról jobbra olvasásnak fontos szerepe van.

A program könnyen átalakítható bármilyen programozási nyelven kóddá, most alakítsuk át pszeudo-kóddá. A programkód úgy generálható, hogy az egyes vezérlési szerkezeteket (jelen esetben csak szekvenciákat) egyenként kifejtjük és annak pszeudo-kódos megfelelőjével helyettesítjük.

```
Eljárás Területszámítás
    Beolvas: A
    Beolvas: B
    Beolvas: C
    S=(A+B+C)/2
    T=sqrt(S*(S-A)*(S-B)*(S-C))
    Kiír: „A terület:”
    Kiír: T
Eljárás vége
```

Jelen esetben a Jackson-ábra felső szintje egy háromelemű szekvencia, amely egyáltalán nem jelenik meg a pszeudo-kódban (pl. nincs olyan programsor, hogy „adatok beolvasása”): ezeket az elemeket tovább fejtettük és egy egyszerű szekvenciális programot készítettünk, amiből hiányzik a Jackson-ábra hierarchiája, illetve hiányoznak a Jackson-ábra közbülső hierarchia-szintjei. Ezzel a kérdéskörrel majd az eljárások és függvények kapcsán részletesebben foglalkozunk.

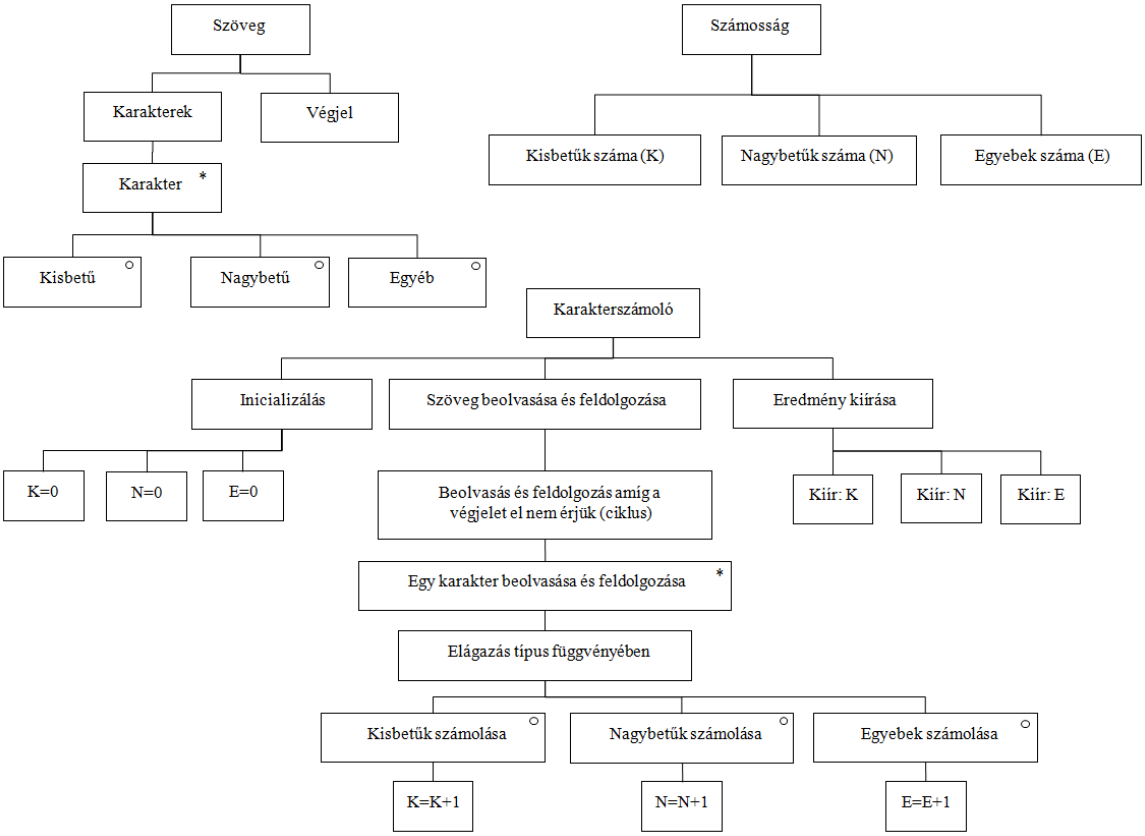
5.3. Példa: Egy szöveg kis- és nagybetűket, valamint egyéb karaktereket (szóköz, írásjelek, stb.) tartalmaz. A szöveg végét egy végjel jelzi. Számoljuk meg a szövegben a kisbetűk, a nagybetűk és az egyéb karakterek darabszámát.

A szöveg leírása az 5.7. ábrán látható: a szöveg sok karakterből áll, amelyek lehetnek kisbetűk, nagybetűk és egyéb karakterek. A programunkban ezek számosságára vagyunk kíváncsiak, ezért a három karaktertípushoz egy-egy számlálót rendelünk (ez lesz a programunk kimenete), ennek leírását szintén az 5.7. ábrán mutatja.

A program a három típusú karakter számára egy-egy számlálót tartalmaz (K, N, E), amelyeket az inicializálás során lenullázunk. Ezután a program egyenként beolvassa a szövegben található karaktereket egészen addig, amíg a végjelet el nem érjük, majd ezen karakterek típusa alapján valamelyik számlálót növeli. Végül a program kiírja mindhárom típusú karakter számosságát.

A program struktúrájában ismét felfedezhetünk hasonlóságokat a bemeneti és kimeneti adatszerkezetekkel, ami nem véletlen: a bemenetet annak szerkezete szerint kell beolvasni és feldolgozni, a kimenetet pedig szintén annak struktúrája szerint kell kiírni. A feldolgozás jelen esetben abból áll, hogy az éppen beolvasott karakter típusának megfelelő számlálót növeljük.

A program pszeudo-kódos leírása pontosan tükrözi a Jackson-ábrán megtervezett programszerkezetet: az egyes tevékenységek és tevékenységszerkezetek kölcsönösen egyértelmű



5.7. ábra. A karakterszámoló program bemeneti és kimeneti adatszerkezeteinek, valamint a program vezérlési szerkezetének leírása Jackson-ábrák segítségével.

megfelelőjét megtaláljuk a pszeudo-kódban.

```
Eljárás Karakterszámoló
    K:=0
    N:=0
    E:=0
    Ciklus
        Beolvas: C
        Ha C kisbetű akkor
            K:=K+1
        Különben ha C nagybetű akkor
            N:=N+1
        Különben ha C egyéb karakter (de nem végjel) akkor
            E:=E+1
        Elágazás vége
    Mígnem C végjel
    Kiír: K
    Kiír: N
    Kiír: E
Eljárás vége
```

Megjegyzés: amikor a Jackson-ábrát vezérlési szerkezetek leírására használjuk, az ábra célja az, hogy a program szerkezetét leírjuk, megtervezzük, így sokszor az apró részletek jelölésével itt nem bajlódunk. Példánkban például az elágazást pontosabban is jelölhettük volna (pl. akkor növeljük K értékét, ha a karakter kisbetű, akkor növeljük N-t, ha a karakter nagybetű, stb.), de a jelen leírás is pontosan érthető. Hasonlóan a ciklus leírása a Jackson-ábrán a magas szintű célt jelzi: addig kell a feldolgozást végezni, amíg a végjelet el nem érjük. Ennek pontos megfogalmazását (elöl vagy hátultesztelő ciklust alkalmazzunk, bennmaradási vagy kilépési feltételt használjuk) általában a Jackson-ábrán nem jelöljük, ezt a programkód (vagy pszeudokód) kidolgozásánál végezzük el (hiszen ez akár programozási nyelvtől is függhet).

5.3. Reguláris kifejezések és definíciók

A reguláris kifejezések a számítástechnikában igen széles körben elterjedtek, segítségükkel tömör formában írhatunk le általános mintázatokat. Az alábbi egyszerű példák néhány tipikus jelölésmódot illusztrálnak.

- alma – ez a reguláris kifejezés csak az „alma” karaktersorozatra illik.
- alma|körte – ez a kifejezés illeszkedik az alma vagy a körte karaktersorozatokra.
- l(é|á)gy – a légy vagy a lágy szavakra illeszkedik
- hahó* – illeszkedik a hah, a hahó, a hahóó, a hahóóó, stb. karaktersorozatokra.
- hahó+ – illeszkedik a hahó, a hahóó, a hahóóó, stb. karaktersorozatokra (de a hah-ra nem).
- (an)?alfabéta – ez lehet analfabéta vagy alfabéta.
- (kis|vén)*asszony – illeszkedik pl. a következő karaktersorozatokra: asszony, kisasszony, vénasszony, kiskisasszony, kiskiskisasszony, vénvénvénvénasszony, kisvénkiskisvénkisasszony, stb.

A fenti reguláris kifejezések tartalmaznak szó szerinti (literális) illesztést (pl. az alma karaktersorozatban valamennyi karakterre pontosan kell illeszkedni), alternatívákat (alma vagy körte), ismétléseket, (tetszőleges számú ó a hahó-ban), opcionális előfordulást (az an karaktersorozat vagy előfordul, vagy nem az alfabéta előtt), illetve ezek kombinációit. A reguláris kifejezések ennél még sokkal több operátorral rendelkeznek, amelyekkel bonyolult mintázatokat egyszerűen és tömören lehet leírni; a számunkra szükséges eszközkészlet csak a fenti szűkített részhalmaz lesz. A reguláris kifejezések következő operátorait fogjuk használni:

- Választás: a | szimbólum az előtte és utána álló alternatívák közül az egyiket választja ki.
- Mennyiség jelzése:
  - A * operátor az előtte álló kifejezésből 0, 1, 2, stb. számú előfordulást engedélyez.
  - A + operátor az előtte álló kifejezésből 1, 2, stb. számú előfordulást engedélyez.
  - A ? operátor 0 vagy 1 előfordulást engedélyez.
- Csoportosítás: A zárójel segítségével szabályozhatjuk az operátorok hatáskörét.

Reguláris kifejezésekhez neveket rendelhetünk, így ennek segítségével további fogalmakat is létrehozhatunk; ennek eszközei a reguláris definíciók. Pl. a számjegy fogalmát így definiálhatjuk:

számjegy → 0|1|2|3|4|5|6|7|8|9

Az általunk definiált számjegy jelentése a mögötte álló reguláris kifejezésből adódóan: vagy egy 0 számjegy, vagy egy 1 számjegy, stb.

Ezt a fogalmat már használhatjuk további reguláris definíciók létrehozásához is. Definiáljuk pl. az egész szám fogalmát a következőképpen:

egész_szám → számjegy(számjegy)* (vagy egyszerűbben: egész_szám → számjegy+)

Ez a definíció tetszőleges (de legalább 1) számjegyből álló számokat megenged. A definíciónk szerint tehát számok például: 5, 314, 19291024, 0, 00000, 000678872.

A reguláris definíció formája tehát a következő:

definiálandó fogalom (név) → *reguláris kifejezés*

A definíció a bal oldalon álló névhez rendeli hozzá a jobb oldalon áll a reguláris kifejezést. Egy definiált név a következő reguláris definícióban már szerepelhet.

5.4. Példa: Definiáljuk az 5.3. ábrán látható egyszerű könyv fogalmát reguláris kifejezések és definíciók segítségével. (Feltételezzük, hogy a szó, oldalszám, címlap, hátlap fogalmakat már korábban definiáltuk, vagy ezek az olvasó számára már ismert fogalmak.)

Ez egy részletes leírás:

szavak → szó*

sor → szavak

sorok → sor*

oldal → sorok oldalszám

oldalak → oldal*

könyv → címlap oldalak hátlap

Ez egy tömörebb leírás, ahol nem definiáltunk minden köztes fogalmat, (pl. kihagyjuk a szavak definícióját):

sor → szó*

oldal → sor* oldalszám

könyv → címlap oldal* hátlap

Ez pedig egy nagyon tömör leírás, egyetlen köztes fogalmat sem használva:

könyv → címlap (szó* oldalszám)* hátlap

A Jackson-ábra átírása reguláris kifejezéssé meglehetősen egyszerű: az ábrán szereplő összes fogalom definícióját mechanikusan helyettesítjük annak reguláris kifejezés párjával: az iteráció helyett a (mag)* jelölést alkalmazzuk, az elágazás helyett az $(ág_1)|(ág_2)|...|(ág_n)$ jelölést, míg a szekvencia helyett pedig az egyszerű felsorolást.

A reguláris kifejezések átírása Jackson-ábrává több körültekintést igényel, hiszen a Jackson-ábra egy fogalma vagy csak szekvenciaként, vagy csak iterációként, vagy csak szelekcióként definiálható: itt szükség szerint újabb fogalmak bevezetése lehet szükséges (pl. a könyv esetén az „oldalak” fogalomra szükség van, hogy a címlap – oldalak – hátlap szekvencia létrehozható legyen és ne az 5.4. ábra szerinti szintaktikai hibás megoldásra jussunk).

Feladatok:

5.1. Rajzoljuk fel a leckekönyv leírását, az alábbi módosításokkal:

Egy tantárgyhoz tartozhat előadás és gyakorlat is (egyszerre is)

A tantárgyak előadásainak és gyakorlatainak óraszámát és meg kell adni (pl. egy tárgy lehet előadás heti 4 órában, a másik gyakorlat heti 2 két órában, egy harmadik heti két előadást és két gyakorlatot tartalmaz).

A tantárgyakhoz rendeljünk oktatót (minden tárgyhoz egyet).

Az oktatóknak legyen neve.

Az oktatók és hallgatók nevét definiáljuk tovább: a név vezetéknévből és egy vagy két keresztnévből álljon.

Egészítsük ki a tantárgyat egy aláírás rovattal, amelynek értéke lehet „aláírva”, „megtagadva”, vagy „nincs kitöltve”. Az osztályzat lehet „elégtelen”, „elégséges”, „közepes”, „jó”, „jeles”, vagy „nincs kitöltve”.

5.2. Készítsük el a területszámító program Jackson-ábrás leírását arra az esetre, amikor a háromszög két oldalát (A és B), valamint az általuk közbezárt szöget (γ) tároljuk. (A terület ekkor a $T = \frac{AB \sin\gamma}{2}$ képlettel számítható.)

A programot készítsük el oly módon is, hogy a szöget fokban olvassuk be, de azt a szinusz függvény számításához szükséges radiánba átalakítjuk ($\gamma_{rad} = \frac{\pi}{180}\gamma_{fok}$).

Módosítsuk a programot úgy, hogy a bemenő adatok beolvasása előtt írja ki, hogy milyen adatot vár (pl.: „Kérem az A oldal értékét:”).

5.3. Javítsunk az egész számokat definiáló reguláris definíciónkon úgy, hogy a számok ne kezdődhessenek nullával – kivéve magát a 0 számot.

5.4. A változók neve egy programozási nyelvben olyan karaktersorozat lehet, ami betűvel kezdődik és utána tetszőleges számú betűt és számot tartalmazhat (a betű nem lehet ékezetes). Pl. legális változónevek: ALMA12, tmp, i, ZiZi12. Nem legális változónevek pl. a következők: 12, 4musketas, pálpusztaisajt, macska:nyelv, stb. Definiáljuk a változónév fogalmát reguláris definíciókkal a fentiek szerint.

5.5. Az alábbi definíciók egy egyszerű, fORDÍT nevű programozási nyelv használatát mutatják be. A definíció alapján rajzoljuk fel a fORDÍT program Jackson-ábrás megfelelőjét.

fORDÍT program → Fejléc Programtörzs Búcsú

Fejléc → „fORDÍT” Program_neve „start”

Búcsú → „TÍDROf”

Programtörzs → Változók_deklarálása Utasítások

Változók_deklarálása → „USING” Változók „GNISU”

Változók → Változó*

Utasítások → Utasítás*

Utasítás → Beolvasás | Kiírás | Művelet | Ciklus

Beolvasás → „READ” változó

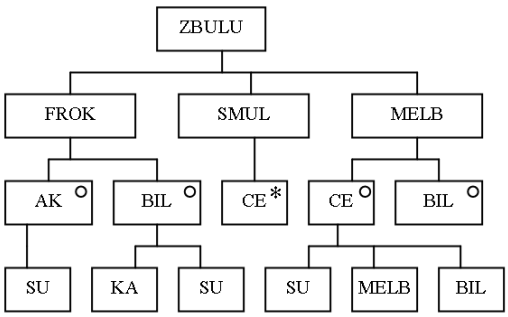
Kiírás → „PRINT” változó
Művelet → Argumentum1 Argumentum2 Operátor Eredmény
Argumentum1 → Argumentum
Argumentum2 → Argumentum
Argumentum → Változó | Konstans
Eredmény → Változó
Operátor → „PLUS” | „MINUS” | „MULT” | „DIV” // összeás, kivonás, szorzás, osztás
Ciklus → „CIKLUS” Hányszor Mag „SULKIC”
Hányszor → Argumentum
Mag → Utasítások

5.6. A fenti fORDÍT programozási nyelven készítsünk programot, ami
a. beolvas egy számot és kiírja azt
b. beolvas egy számot és kiírja annak kétszeresét
c. beolvas három számot és kiírja azok átlagát
d. beolvas ezer számot és kiírja azok átlagát
e. kiírja a 2011 számot.

5.7. Rajzoljuk fel Jackson-ábra segítségével a vállalat szerkezetét. A *vállalat* egy *titkárság*ból, egy *marketing osztály*ból, valamint sok *fejlesztési osztály*ból áll. Minden osztályon és titkárságon vannak *bútor*ok (ami *szék*, *asztal*, *fotel* lehet), sok *számítógép* (ami *PC* vagy *szerver* lehet), valamint egy *kávéfőző* vagy egy *vízmelegítő* (de csak az egyik!). Az ábrán az összes dőlt betűkkel szedett szónak szerepelnie kell (de szükség szerint más szavak is szerepelhetnek).

5.8. Rajzoljuk fel Jackson-ábra segítségével az ország szerkezetét. Az ábrán a következő elemeknek szerepelni kell: *Ország, város, utca, lakóház, megye, megyeszékhely, falu, polgármesteri hivatal, iskola, templom, bolt*.

5.9. Az alábbi Jackson-ábra alapján válaszoljunk a feltett kérdésekre! Az „akármilyen sok” jelzésére írjunk X-et a táblázatba.



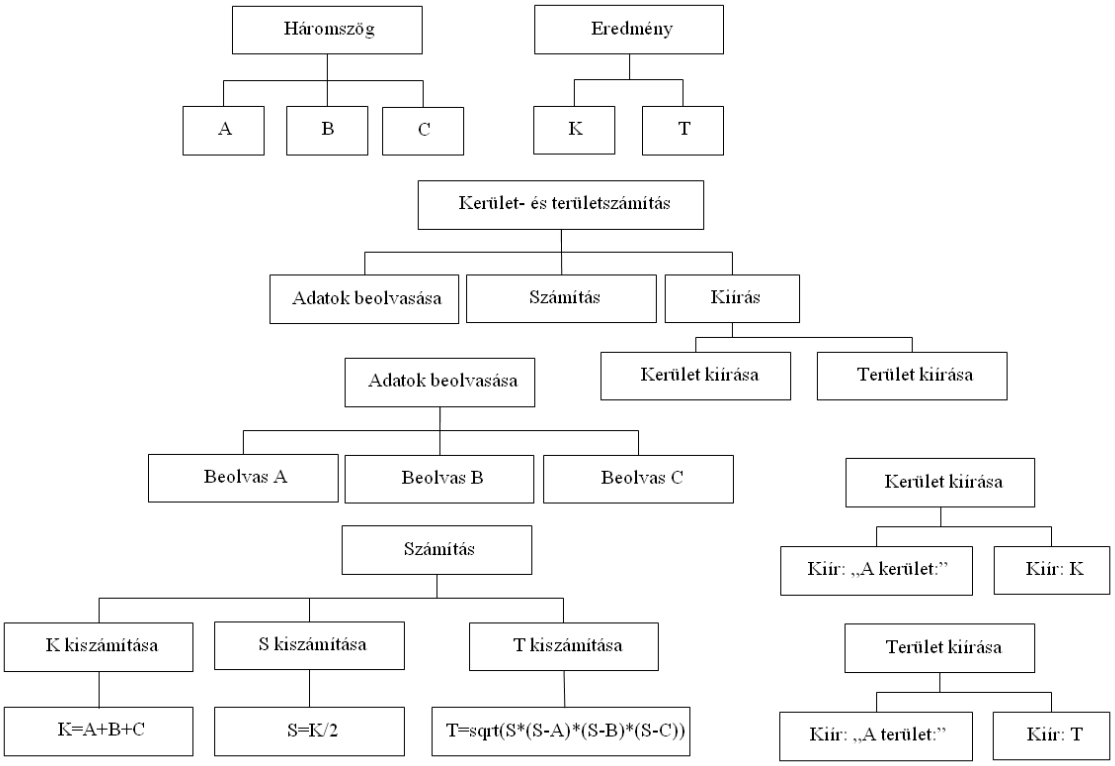
	MIN	MAX
Hány SU van a ZBULU-ban?		
Hány CE van a ZBULU-ban?		
Hány KA van a ZBULU-ban?		
Hány BIL van a ZBULU-ban?		
Hány MELB van a ZBULU-ban?		
Hány SMUL van a ZBULU-ban?		
Hány BIL van a MELB-ben?		
Hány CE van a SMUL-ban?		

6. fejezet

Szekvenciális adat- és programszerkezetek

A szekvenciális szerkezetek a legegyszerűbb előforduló szerkezetek programjainkban. Szekvenciális vezérlési szerkezettel sok egyszerű feladat megoldható, mind pl. a háromszög területének számítása, amelynek Jackson-ábráját az 5.6. ábra mutatja. A területszámító program először beolvassa egymás után a háromszög oldalainak hosszát, majd kiszámítja a fél kerület értékét (ez az S változó), majd a terület kiszámítása következik, végül két egymás utáni kiírást végez el. A programot egy egyszerű szekvenciaként implementálhatjuk.

6.1. példa: Az 5.2. példában tárgyalt területszámító program mintájára készítsük el azt a programot, amelyik kiszámítja a háromszög oldalaiból a háromszög kerületét és területét. A megoldás Jackson-ábrája az 5.6 ábrához hasonlóan alakul, a módosított ábra a 6.1. ábrán



6.1. ábra. A kerület- és területszámító program bemeneti és kimeneti adatszerkezeteinek, valamint a program vezérlési szerkezetének leírása Jackson-ábrák segítségével.

látható. Ezen az ábrán a bemeneti és kimeneti adatok szerkezetét is feltüntettük.
A program pszeudo-kódja a következő:

```
    Eljárás Kerület- és területszámítás
        Beolvas: A
        Beolvas: B
        Beolvas: C
        K=A+B+C
        S=K/2
        T=sqrt(S*(S-A)*(S-B)*(S-C))
        Kiír: „A kerület:”
        Kiír: K
        Kiír: „A terület:”
        Kiír: T
    Eljárás vége
```

Az algoritmust valamilyen valós programozási nyelven is implementálhatjuk. A kód C nyelven a következő lehet [*6.heron.c*]:

```
/*
* Háromszög területének meghatározása Héron képlete alapján.
*/

#include <stdio.h>   /* printf, scanf miatt */
#include <math.h>    /* sqrt miatt */

double A, B, C, T, S, K;

int main(){
     printf("Az A oldal hossza:");
     scanf("%lf", &A);
     printf("A B oldal hossza:");
     scanf("%lf", &B);
     printf("A C oldal hossza:");
     scanf("%lf", &C);
     K=(A+B+C);
     S=K/2;
     T=sqrt(S*(S-A)*(S-B)*(S-C));
     printf("\nA terület: %lf\n", T);
     printf("\nA kerület: %lf\n", K);
     return 0;  /* sikeres végrehajtás jelzése */
}
```

A fenti kód egy megjegyzéssel indul. A C nyelvben megjegyzéseket a /* nyitó és */ záró szekvencia közé lehet helyezni (akár több soron keresztül is), vagy a // szekvencia után (csak egy sornyi).

A kód érdemi része néhány gyakran használatos C függvényt tartalmazó könyvtár beillesztésével indul. Az #include direktívák jelzik a fordító program számára, hogy használja a <> szimbólumok között megadott állományokban található függvényeket. Az stdio.h például a később használt printf() és scanf() függvények, míg a math.h a sqrt() függvény deklarációját tartalmazza. Mivel a C programozási nyelvnek nem részei még az ilyen egyszerű függvények sem, az #include direktívák hiányában a fordító hibát jelezne, hiszen nem tudná, mit jelent pl. az sqrt() függvény. (Néhány fejlesztői környezet már olyan „okos”, hogy ezen direktívák nélkül is megtalálja a könyvtárakban levő standard függvényeket, de erre ne számítsunk.)

Szigorú típusellenőrző nyelvekben – a C nyelv is ilyen – a változókat és azok típusát az első használat előtt deklarálni kell. A deklaráció a fordító számára jelzi, hogy a változónak mi a típusa és a neve. Ezen kívül a változókat definiálni is kell, amikor is a változó valóban létrejön, a tárban számára hely foglalódik. A C nyelvben a változók esetén ez a két fogalom egybemosódik, mert az esetek többségében a változókat egyszerre deklaráljuk és definiáljuk is. (Ez alól egyetlen kivétel van, amikor egy változót több modulban is használni akarunk, ekkor azt egy helyen definiáljuk, de több modulban is deklarálhatjuk. A külső változó deklarációjára az `extern` kulcsszóval történik, de ennek részleteivel itt nem foglalkozunk.) A deklaráció és definíció közötti különbséggel a függvények kezelése kapcsán a 9.1. fejezetben találkozunk majd.

Példánkban az összes változó double (dupla pontosságú lebegőpontos szám) típusú, jelen esetben a main függvény előtt deklaráltuk (és definiáltuk) őket. A C nyelvben a deklarációt a típus nyitja, majd a változók nevei következnek vesszővel elválasztva. A deklarációt a pontosvessző zárja. Egyéb deklarációkra példák:

```
int i, j;     /* egészek */
char c;       /* karakter */
char str[12]; /* 12 elemű karaktertömb (karaktersorozat) */
```

Minden C nyelven írott programok végrehajtása a `main` függvényben kezdődik. A függvényekkel a későbbiekben foglalkozunk, most csak ezt az egyetlen különleges függvényt kell használnunk. Ezen függvény egy egész számot ad vissza, ahogyan azt az int típus jelzi (általában a 0-t, ami megállapodás szerint a hibátlan végrehajtást jelzi) és nincs bemenő paramétere, ahogy az üres zárójel mutatja a függvény neve mögött. A függvény törzse a kapcsos zárójel-pár között található.

A `printf` függvény használható igen változatos formában a kiírások végrehajtására. A `printf("Az A oldal hossza:")` utasítás egy karaktersorozatot ír ki a képernyőre, a függvény bemenő paramétere a kiírandó karaktersorozat maga. A `printf("\nA terület: %lf\n", T)` függvény egy haladóbb alkalmazása a kiíró függvénynek. Az első paraméter egy vezérlő mező, amelyben a nyomtatható karakterek mellett vezérlő szekvenciákat is találhatunk, jelen esetben a `%lf` formátumvezérlőt és a `\n` karaktersorozatot. A `%lf` formátumvezérlőt jelentése az, hogy erre a helyre egy lebegőpontos számot kell írni, míg a `\n` egy sortörést helyez el az adott helyre. (Gyakran használatosak még a `%d`, `%c` és `%s` formátumvezérlők, amelyek jelenése rendre: egy egész szám, karakter és karaktersorozat.) A kiírandó lebegőpontos szám értékét a printf() függvény második paramétereként adjuk át, jelen esetben

ez a T lebegőpontos szám. A fenti parancs tehát először sort emel, majd kiírja az „A terület: ” karaktersorozatot, majd a T lebegőpontos szám értékét, amit egy újabb soremelés követ.

A printf függvény vezérlő mezőjében tetszőleges számú formátumvezérlő helyezhető el, természetesen ennek megfelelő számú paramétert kell a formátumvezérlő mező után még átadni a printf() függvénynek. Pl. a `printf(" Ide jön egy egész: %d, ide egy karakter: %c, ide meg egy karaktersorozat: %s ", 3, 'Q', "ALMA")` függvényhívás a következő szöveget írja ki:
Ide jön egy egész: 3, ide egy karakter: Q, ide meg egy karaktersorozat: ALMA.
A printf függvény számos egyéb lehetőséggel rendelkezik, ezek részletes leírása megtalálható a [4] irodalomban. A leggyakrabban használt formátumvezérlők ismertetése az F2. függelékben található.

A C nyelvben a scanf() függvény szolgál formázott beolvasásra. Első paramétere a formátumvezérlő karaktersorozat, a többi paramétere a beolvasandó változók címeit tartalmazza. Például a `scanf("%lf", &A)` parancsban a formátumvezérlő egy darab lebegőpontos szám beolvasását kéri, amelyet az A változóba kell tölteni, amit a függvény második paramétere jelez: a & címképző operátorral jelezzük, hogy nem az A változó értékét adjuk át a függvénynek, hanem a változó címét. A printf() függvényhez hasonlóan más vezérlő karakterek is használhatók, így például a `scanf("%d", &X)` parancs egy egész számot olvas be az X változóba, a `scanf("%c", &c)` paranccsal egy karakter olvasunk be a c változóba, míg a `scanf("%s", str)` parancs egy karakterláncot olvas be a str nevű változóba. (Vigyázat: a C nyelvben a karaktertömbök neve a változó címét (és nem értékét) jelzi, így az utolsó példában a `str` változó neve elé már nem kell a címképző operátor.)

A C nyelvben a matematikai műveletek jelölése a „szokásos” módon történik, az értékadást az egyenlőségjel jelzi. A négyzetgyök számítására használatos függvény a `sqrt`.

A programot a `return` utasítás zárja, amely a `main` függvény visszatérési értékét állítja be nullára, amely a programot hívó számára jelzi, hogy a végrehajtás rendben lezajlott.

A C nyelvben az utasításokat pontosvesszővel kell lezárni.

A fenti megoldásban a kód szerkezete tükrözi a Jackson-ábrán látható szerkezetet, ráismerünk az ábrán látható szekvenciákra. A programban használt változók azonban nem tükrözik a köztük lévő viszonyokat: a háromszög oldalai az A, B, C változókban vannak tárolva, de ezekről nem derül ki, hogy egymással kapcsolatban vannak (nevezetesen egyazon háromszög oldalai). Ugyanez igaz a kerületre és a kerületre is: a K és T nevű változók egymástól függetlenek, ezek viszonya sem tükröződik a program adatszerkezetében.

A Jackson-ábrán látható, hogy a Háromszög nevű entitás az A, B, és C nevű entitások szekvenciája. Hasonlóan megjelenik az ábrán az Eredmény nevű entitás is, ami a K és T entitások szekvenciája. Ezek a fogalmak nem jelennek meg a programban. Módosítsuk tehát programunkat úgy, hogy annak adatszerkezete is tükrözze a valóságban köztük lévő viszonyokat. Ennek eszköze a programozási nyelvekben a struktúra.

A struktúra logikailag összetartozó, akár különféle típusú adatok tárolására alkalmas. Példánkban egy struktúra lehet a Háromszög, amely három adattagot (vagy más néven mezőt) tartalmaz. Ezekre úgy szokás hivatkozni, hogy megadjuk a változó (a struktúra) nevét, majd a pont operátor után megadjuk a hivatkozott mező nevét is. Példánkban a háromszög oldalaira így hivatkozhatunk: `Háromszög.A`, `Háromszög.B` és `Háromszög.C`. hasonlóan az eredmény struktúra két mezője a kerület és terület, amelyeket rendre a `Eredmény.K` és `Eredmény.T` hivatkozásokkal érhetjük el.

A kerület- és területszámító algoritmus pszeudo-kódja struktúrákkal a következő lehet:

```
Eljárás Kerület- és területszámítás (struktúrákkal)
   Beolvas: Háromszög.A
   Beolvas: Háromszög.B
   Beolvas: Háromszög.C
   Eredmény.K=Háromszög.A+Háromszög.B+Háromszög.C
   S=K/2
   Eredmény.T=sqrt(S*(S-Háromszög.A)*(S-Háromszög.B)*(S-
   Háromszög.C))
   Kiír: „A kerület:”
   Kiír: Eredmény.K
   Kiír: „A terület:”
   Kiír: Eredmény.T
Eljárás vége
```

A kódban látható, hogy csupán egyetlen, a könnyebb áttekinthetőség érdekében bevezetett átmeneti változónk van (S), a Háromszög és az Eredmény változók most már tartalmazzák a korábban különálló életet élő, de logikailag összefüggő változókat. A program látszólag bonyolultabb lett, hiszen a különálló változókra rövidebben lehet hivatkozni, mint a struktúra tagjaira, de ez a kis kényelmetlenség a továbbiakban bonyolultabb programoknál bőségesen megtérül azzal, hogy az összetartozó adattagokat elegánsan, együtt kezeljük (például egyetlen értékadással adhatunk értéket egy struktúrának). Ezekkel a kérdésekkel bővebben a 10. fejezetben foglalkozunk.

6.2. példa: Készítsük el a struktúrákkal megvalósított program C nyelvű változatát.

A program a következő [*6.haromszog_s.c*]:

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

struct haromszog{
   double A;     /* a oldal hossza */
   double B;     /* b oldal hossza */
   double C;     /* c oldal hossza */
};

struct eredmeny{
   double K;     /* kerület */
   double T;     /* terület */
};

struct haromszog Haromszog;  /* bemeneti adatok */
```

```
struct eredmeny Eredmeny;    /* végeredmény */
double S;

int main(){
   /* Az A oldal bekérése: */
   printf("Az A oldal hossza:");
   scanf("%lf", &Haromszog.A);

   /* A B oldal bekérése: */
   printf("A B oldal hossza:");
   scanf("%lf", &Haromszog.B);

   /* A C oldal bekérése: */
   printf("A C oldal hossza:");
   scanf("%lf", &Haromszog.C);

   Eredmeny.K=(Haromszog.A+Haromszog.B+Haromszog.C);  /* kerület */
   S=Eredmeny.K/2;     /* a fél kerület Héron alábbi képletéhez */
   Eredmeny.T=sqrt(S*(S-Haromszog.A)*(S-Haromszog.B)*(S-
Haromszog.C));

   /* Az eredmények kiírása: */
   printf("\nA terulet: %lf\n", Eredmeny.T);
   printf("\nA kerulet: %lf\n", Eredmeny.K);
   return 0;
}
```

A struktúrák új típusokat hoznak létre a beépített típusok (pl. double, int) mellé. Ezeket a típusokat először definiálni kell. A fenti példában létrehozunk két új struktúra típust, amelyeket `haromszog` és `eredmeny` neveken definiálunk. A `haromszog` struktúra típus mezőinek neve rendre A, B és C, a mezők mindegyike double típusú. A struktúrák definíciója a `struct` kulcsszóval kezdődik, majd a struktúra típus neve következik. Ezután kapcsos zárójelek között megadjuk egyenként a mezők típusát és nevét pontosvesszővel lezárva (hasonlóan a változódeklarációhoz). A struktúra definícióját pontosvessző zárja.

A `haromszog` struktúra típushoz hasonlóan definiáljuk az `eredmeny` struktúratípust is, amelynek két mezője a K és T és ezek szintén `double` típusok.

Figyelem: a `haromszog` és `eredmeny` még nem változók, ezek csak változó típusokat definiálnak. Ezek segítségével definiálhatunk olyan változókat, amelynek szerkezete megegyezik a típusdefinícióban leírtakkal. A struktúrák létrehozása a program következő két sorában történik. A változó deklarációk szabályai szerint a típust követi a változó neve: jelen esetben a típus `struct haromszog`, míg a változó neve `Haromszog` (a C nyelv megkülönbözteti a kis- és nagybetűket). Hasonlóan a `struct eredmeny` típusból létrehozunk egy `Eredmeny`

nevű változót. (Figyelem: a C nyelvben a struktúrák típusa elé mindig oda kell írni a struct kulcsszót is!)

A struktúra mezőire a változó nevével, majd a pont operátor után a mező nevével hivatkoznunk. Pl. a Haromszog.C a Haromszog nevű struktúra C nevű mezőjét jelenti.

Figyelem: csakúgy, mint bármely változónak, a struktúra mezőjének is van típusa. Jelen példánkban mindkét struktúra valamennyi mezője double típusú volt.

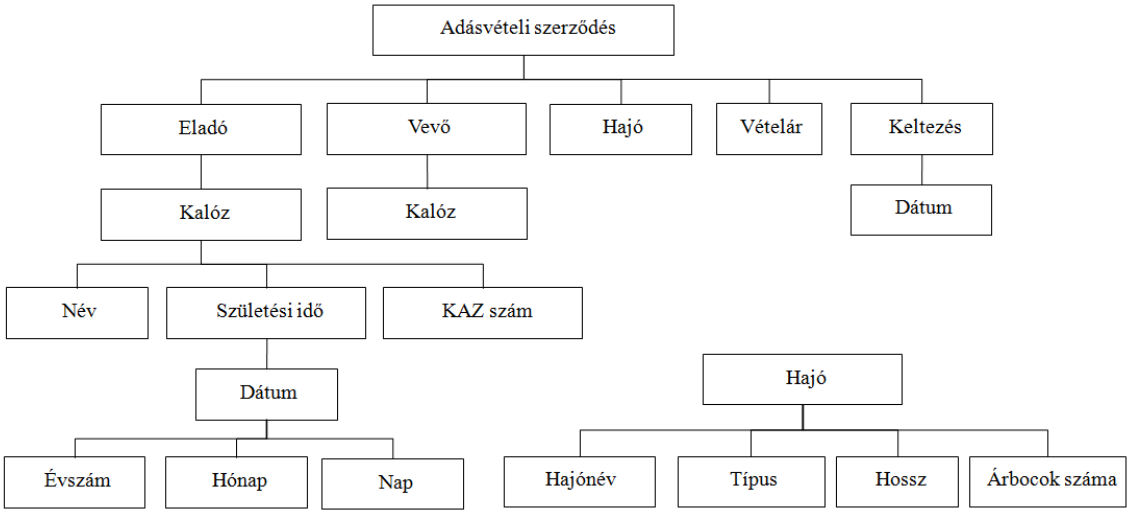
A példában látható módon a struktúra mezőinek ugyanúgy adunk értéket, mint a hasonló típusú változóknak. A hivatkozott mezőket pedig ugyanúgy használhatjuk (pl. aritmetikai operációkban), mint a hasonló típusú változókat.

És végül egy szomorú hír: a C nyelvben a változók, típusok, (s később a függvények) nevei sajnos nem tartalmazhatnak ékezetes karaktereket. Ezért használtunk pl. `Eredmeny` nevű változót a szebb – de hibás –Eredmény helyett.

A kalózok nagy élvezettel és gyakran adják-veszik egymás között a zsákmányolt hajókat. Sajnos a rum hatása alatt köttetett szóbeli megállapodásokra másnap már gyakran nem emlékeznek a felek, ami sokszor komoly, sőt vérre menő vitákat eredményez. Morc Misi ezért korszerűsíteni akarja a hajó-adásvételek adminisztrációját: számítógépes programot akar készíttetni e célra, amellyel szép formátumban lehet hivatalos szerződéseket gyártani. Egy hajó adásvételi szerződése tartalmazza az eladó, a vevő, valamint a hajó adatait, valamint a vételárat és az adásvétel dátumát. Az eladóról és a vevőről tároljuk a nevét, születési idejét és a titkos kalóz-azonosító (KAZ) számát, a hajóról pedig a nevét, típusát, hosszát és árbocszámát.

6.3. példa: Írjunk egy egyszerű programot Morc Misinek, amely beolvassa egy szerződés adatait, majd kiírja azokat a standard kimenetre (ez általában a képernyő). (Innen Misi tudós alárendeltjei már tudnak papírra is nyomtatni) [*6.adasvetel1.c*].

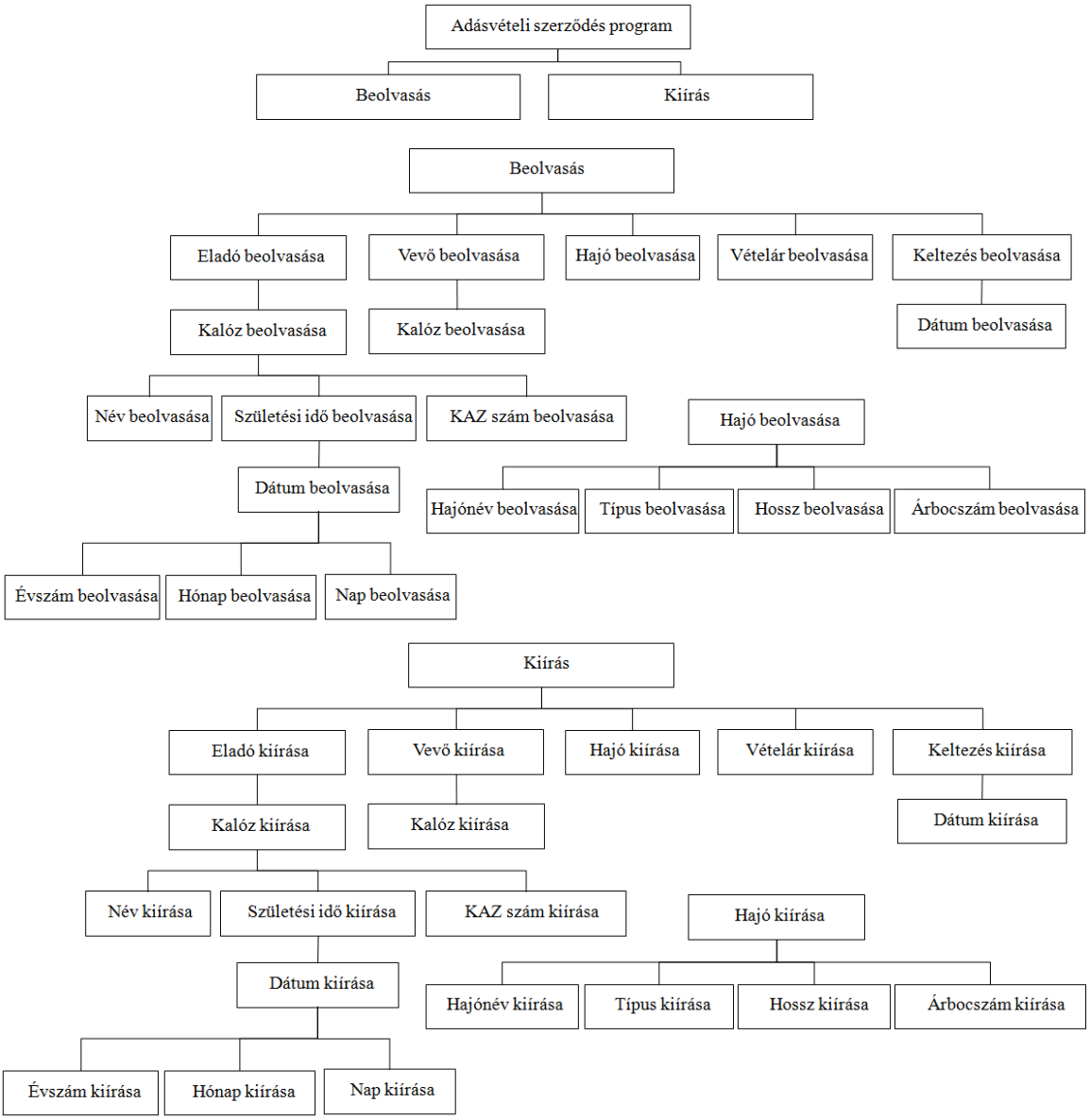
A 6.2. ábra mutatja a feladat megfogalmazása szerinti adatszerkezeteket. Az adásvételi szerződés szekvenciaként tartalmazza az eladó adatokat, a vevő adatokat, a hajó adatait, a vételárat és a keltezést. Szintén szekvenciaként ábrázolhatók a vevő és eladó adatai, ráadásul ezek ugyanolyan szerkezetűek, hiszen kalózokat írnak le: tartalmazzák a nevüket, születési dátumukat és a titkos KAZ-számukat. A hajó adatai szintén szekvenciát alkotnak, melynek



6.2. ábra. A hajó adásvételi szerződés adatszerkezetének leírása Jackson-ábra segítségével.

elemei a hajó neve, típusa, hossza és árbocainak száma. Megfigyelhetjük továbbá, hogy a dátum több helyen is szerepel: mind a szerződés keltezése, mind a születési idők dátumok. A feladat megfogalmazása ugyan nem tartalmazza, de még a hajósinasok is tudják, hogy a dátum egy évszámból, egy hónapból és egy napból áll (ami ismét egy szekvencia).

A program szerkezete nagyon egyszerű, ahogyan az a 6.3. ábrán látható: egy beolvasást és egy kiírást tartalmaz, ahol mindkét tevékenység pontosan tükrözi az adatszerkezetet (hiszen mind a beolvasás, mind a kiírás pontosan ezt az adatszerkezetet járja be).



6.3. ábra. A hajók adásvételi szerződését kezelő program leírása Jackson-ábra segítségével.

A 6.2. ábra szerinti adatszerkezet leírására ismét a struktúrák adnak természetes lehetőséget. Az ábra szerint egy struktúrában ábrázolhatjuk magát a Szerződést, amelynek mezői az Eladó, Vevő, Hajó, Vételár, Keltezés lesznek. Az eladót és a vevőt szintén egy-egy struktúra írja le, melynek mezői a Név, a Születési idő és a KAZ-szám. A hajó is struktúrával ábrázolható, amely a Név, Típus, Hossz és Árbocszám mezőket tartalmazza. A dátumokat is ábrázolhatjuk egy-egy struktúrával, amelynek mezői az Évszám, a Hónap és a Nap lesznek. Fi-

gyeljük meg, hogy a fenti adatszerkezetben vannak olyan struktúrák, amelyek tartalmaznak struktúrákat (pl. a Szerződés Eladó, Vevő és Keltezés mezői szintén struktúrák). Ez természetes, hiszen a szekvenciák elemei lehetnek elágazások, iterációk, vagy akár szekvenciák is: jelen esetben a szekvenciák további szekvenciákat tartalmaznak, amelyek struktúrákat tartalmazó struktúrákká képződnek le adatszerkezeteinkben.

A C nyelvű programban az előzőekhez hasonlóan először definiáljuk a struktúrák típusait, majd ezekből hozunk létre változókat. A struktúrák típusainak a következő neveket választottunk: `szerzodes`, `kaloz`, `hajo`, `datum`. A struktúrák mezőinek nevei a fenti, de ékezetek nélküli mezőnevekkel egyeznek meg. A programban ezen típusok alapján létrehozzuk és használjuk a `Szerzodes`, `Kaloz`, `Hajo` és `Datum` nevű változókat:

```
/*
 * Hajó adásvételi szerződés adatainak bekérése és kiírása.
 */

#include <stdio.h>

#define _MAX_HOSSZ 20

struct datum{
   unsigned int Evszam;          /* a dátum évszám mezője */
   unsigned int Honap;           /* a dátum hónap mezője */
   unsigned int Nap;             /* a dátum nap mezője */
};

struct kaloz{
   char Nev[_MAX_HOSSZ+1];       /* a kalóz neve */
   struct datum Szuletesi_ido;   /* a kalóz születési ideje */
   unsigned int KAZ_szam;        /* a kalóz KAZ-száma */
};

struct hajo{
   char Nev[_MAX_HOSSZ+1];       /* a hajó neve */
   char Tipus[_MAX_HOSSZ+1];     /* a hajó típusa */
   unsigned int Hossz;           /* a hajó hossza */
   unsigned int Arbocszam;       /* a hajó árbocainak száma */
};

struct szerzodes{
   struct kaloz Elado;           /* az eladó adatai */
   struct kaloz Vevo;            /* a vevő adatai */
```

```
   struct hajo Hajo;            /* a hajó adatai */
   unsigned int Vetelar;        /* a hajó vételára */
   struct datum Keltezes;       /* a szerződés kelte */
};

struct datum Datum;
struct kaloz Kaloz;
struct hajo Hajo;
struct szerzodes Szerzodes;

int main(){
   printf("Kaloz adasveteli szerzodes bevitele\n");

   /* az eladó adatainak bekérése mezőnként a Kaloz segédváltozóba*/
   printf("\n\tElado adatai:\n");
   printf("\t\tNev: ");
   scanf("%s", Kaloz.Nev);
   printf("\t\tSzuletesi ido [e.h.n.]: ");
   scanf("%d.%d.%d.", &Datum.Evszam, &Datum.Honap, &Datum.Nap);
   Kaloz.Szuletesi_ido=Datum;
   printf("\t\tTitkos kaloz azonosito: ");
   scanf("%d", &Kaloz.KAZ_szam);
   Szerzodes.Elado=Kaloz;  /* tárolás a szerződésben */

   /* a vevő adatainak bekérése mezőnként a Kaloz segédváltozóba */
   printf("\n\tVevo adatai:\n");
   printf("\t\tNev: ");
   scanf("%s", Kaloz.Nev);
   printf("\t\tSzuletesi ido [e.h.n.]: ");
   scanf("%d.%d.%d.", &Datum.Evszam, &Datum.Honap, &Datum.Nap);
   Kaloz.Szuletesi_ido=Datum;
   printf("\t\tTitkos kaloz azonosito: ");
   scanf("%d", &Kaloz.KAZ_szam);
   Szerzodes.Vevo=Kaloz; /* tárolás a szerződésben */


   /* a hajó adatainak bekérése mezőnként a Hajo segédváltozóba */
   printf("\n\tHajo adatai:\n");
   printf("\t\tNev: ");
   scanf("%s", Hajo.Nev);
```

```
   printf("\t\tTipus: ");
   scanf("%s", Hajo.Tipus);
   printf("\t\tHajo hossza: ");
   scanf("%d", &Hajo.Hossz);
   printf("\t\tHajo arbocszama: ");
   scanf("%d", &Hajo.Arbocszam);
   Szerzodes.Hajo=Hajo; /* tárolás a szerződésben */

   /* a vételár bekérése és tárolása a szerződésben */
   printf("\n\tHajo vetelara: ");
   scanf("%d", &Szerzodes.Vetelar);

   /* a keltezés bekérése és tárolása a szerződésben */
   printf("\n\tSzerzodeskotes idopontja [e.h.n.]: ");
   scanf("%d.%d.%d.", &Datum.Evszam, &Datum.Honap, &Datum.Nap);
   Szerzodes.Keltezes=Datum;

   printf("\n\nKaloz adasveteli szerzodes adatainak listazasa\n");

   printf("\n\tElado adatai:\n");
   Kaloz=Szerzodes.Elado;
   /* eladó adatainak kiírása mezőnként*/
   printf("\t\tNev: %s\n", Kaloz.Nev);
   Datum=Kaloz.Szuletesi_ido;
   printf("\t\tSzuletesi ido: %d.%d.%d.\n", Datum.Evszam,
      Datum.Honap, Datum.Nap);
   printf("\t\tTitkos kaloz azonosito: %d\n", Kaloz.KAZ_szam);

   printf("\n\tVevo adatai:\n");
   Kaloz=Szerzodes.Vevo;
   /* vevő adatainak kiírása mezőnként*/
   printf("\t\tNev: %s\n", Kaloz.Nev);
   Datum=Kaloz.Szuletesi_ido;
   printf("\t\tSzuletesi ido: %d.%d.%d.\n", Datum.Evszam,
      Datum.Honap, Datum.Nap);
   printf("\t\tTitkos kaloz azonosito: %d\n", Kaloz.KAZ_szam);

   printf("\n\tHajo adatai:\n");
   Hajo=Szerzodes.Hajo;
   /* a hajó adatainak kiírása mezőnként*/
   printf("\t\tNev: %s\n", Hajo.Nev);
```

```
   printf("\t\tTipus: %s\n", Hajo.Tipus);
   printf("\t\tHajo hossza: %d\n", Hajo.Hossz);
   printf("\t\tHajo arbocszama: %d\n", Hajo.Arbocszam);

   /* vételár kiírása */
   printf("\n\tHajo vetelara: %d\n", Szerzodes.Vetelar);

   /* keltezés kiírása */
   Datum=Szerzodes.Keltezes;
   printf("\n\tSzerzodeskotes idopontja %d.%d.%d.", Datum.Evszam,
      Datum.Honap, Datum.Nap);

   return 0;
}
```

A program elején található `#define` direktíva a C nyelvben gyakran használatos, ennek segítségével lehet például konstansokat is létrehozni, ahogy azt programunkban tettük is. A `#define _MAX_HOSSZ 20` jelentése a fordítóprogram számára a következő: a programban található összes „_MAX_HOSSZ” karaktersorozatot a „20” karaktersorozattal kell helyettesíteni. Rossz programozási gyakorlat, ha „bűvös számokat” hagyunk a programunkban (pl. `char Nev[_MAX_HOSSZ+1]` helyett `char Nev[21]`), a konstansokat mindig lássuk el beszédes névvel és azokat használjuk. Ezzel programunk sokkal áttekinthetőbb, érthetőbb lesz és sok munkát és kellemetlenséget spórolhatunk meg magunknak a későbbiekben. (Képzeljük el pl. hogy valamilyen okból meg kell növelnünk a programunkban `_MAX_HOSSZ` értékét 20-ról 40-re: ekkor egyetlen sort kell csak módosítanunk és célunkat elértük. Ha a bűvös 21 számot alkalmaztuk volna a programban, akkor több helyen is módosítani kellene a 21 értékét 41-re. De vajon minden programunkban előforduló 21-est módosítani kell? Jelen programunkban igen, de ez nincs mindig így...)

A program a struktúratípusok definíciójával folytatódik. A struktúrák definíciójánál ügyelni kell a definíciók sorrendjére. Például a `szerzodes` típusú struktúrák egyes mezői struktúrák (`kaloz` és `datum` típusúak), így a `szerzodes` struktúratípus definíciójakor a fordítónak már tudni kell, hogy ezek a típusok léteznek. Ehhez a `kaloz` és a `datum` struktúratípusokat a `szerzodes` struktúratípus előtt kell definiálni. Hasonlóan a `datum` struktúratípus definíciójának meg kell előznie a `kaloz` struktúratípus definícióját.

A `datum` struktúra típus három mezőt tartalmaz (`Evszam`, `Honap`, `Nap`), melyek mindegyike `unsigned int` típusú: ez a C nyelvben az előjel nélküli egész számot jelenti (az `int` lehet negatív szám is, az `unsigned int` nem). A `kaloz` struktúratípus mezői a `Nev` karaktertömb, a `Szuletesi_ido`, ami egy `datum` típusú struktúra, valamint az előjel nélküli egész típusú `KAZ_szam`. A C nyelvben a tömböket úgy deklaráljuk, mint a tömb elemeinek megfelelő típust, csak a változó neve (vagy jelen esetben a mező neve) mögött szögletes zárójelben megadjuk a tömb méretét is. A programban a `char Nev[_MAX_HOSSZ+1]` jelentése tehát az, hogy a `Nev` egy `_MAX_HOSSZ+1` méretű tömb, amelynek elemei karakterek. A tömbök használatával részletesen majd a 8. fejezetben foglalkozunk. Most még egy hasznos tudnivaló: a C nyelvben a karakterláncok végét egy speciális záró karakter (melynek kódja a 0) jelzi, ezért a karaktertömbben ennek egy helyet célszerű fenntartani. Ha a `_MAX_HOSSZ` a nevek maxi-

mális hosszát jelenti, akkor a tömbnek `_MAX_HOSSZ+1` hosszúnak kell lennie, hogy a leghosszabb név is elférjen benne a záró karakterrel együtt.

A `hajo` struktúratípus mezői a `Nev` és a `Tipus` karaktertömbök (mindegyik hossza itt is `_MAX_HOSSZ+1`), valamint a `Hossz` és `Arbocszam` előjel nélküli egészek. A `szerzodes` típusú struktúrák tartalmaznak egy `Elado` és egy `Vevo` nevű mezőt, amelyek `struct kaloz` típusúak, egy `Hajo` nevű mezőt, ami `strukt hajo` típusú, egy előjel nélküli egész típusú `Vetelar` mezőt, valamint egy `Keltezes` nevű mezőt, ami `struct datum` típusú.

A fenti struktúrákból ezután létrehozunk egy-egy változót, amelyek neve rendre `Datum`, `Kaloz`, `Hajo` és `Szerzodes`.

A main függvényben a program először az eladó adatait kéri be oly módon, hogy a felhasználó számára először mindig kiírja a bekérendő adat nevét, majd bekéri az adatot. A név egy karakterlánc, ezért a `scanf` függvényben `%s` formátumvezérlőt használjuk (és nincs `&` címképző operátor a változó neve előtt). A nevet a `Kaloz` struktúra `Nev` mezőjében tároljuk, tehát a bekérés során a `Kaloz.Nev` hivatkozást használjuk, ide fogja a `scanf` függvény betölteni a begépelt nevet (egy záró karakterrel a végén). Figyelem: a `scanf` függvény csak szóközig olvas, így a név nem tartalmazhat szóközt: ezzel a megoldással csak egytagú neveket lehet használni. Más függvények (pl. `fgets`) felhasználásával lehetséges egész sorok kezelése, amelyek már szóközöket is tartalmazhatnak.

A születési idő bekérésénél a formázott bemenet kezelésére látunk példát. A dátumot évszám.hónap.nap. formátumban kérjük be. Ezt a formátumot tartalmazza a `scanf` függvény formátumvezérlő mezője: három egész számot kérünk be, amiket pont karakterek választanak el egymástól. A `scanf` függvény a három egész számot rendre a `Datum` struktúra `Evszam`, `Honap` és `Nap` mezőibe tölti. A `Datum` struktúrát a következő értékadó utasítás (`Kaloz.Szuletesi_ido=Datum`) tölti be a `Kaloz` struktúra `Szuletesi_ido` mezőjébe. Figyeljük meg, hogy itt az egész struktúrát (minden egyes mezőjével) egyetlen utasítással adjuk értékül a másik struktúrának, nem szükséges ezt mezőnként végezni. A KAZ-szám betöltése egész számként történik a már ismert módon. Az eladó adatainak beolvasása után a kalóz struktúrát értékül adjuk a `Szerzodes` struktúra `Elado` mezőjének.

A `printf` utasításokban a már ismert `\n` (új sor) mellett egy újabb vezérlő karakterrel találkozhatunk: a `\t` a tabulátor karaktert jelenti.

A vevő adatait hasonló módon olvassuk be ismét a `Kaloz` változóba, majd onnan a `Szerzodes` struktúra `Vevo` mezőjébe.

Az eladó és vevő adatait alternatív módon, a `Kaloz` változó használata nélkül is be lehet olvasni. Ekkor minden adatot közvetlenül a `Szerzodes` struktúra `Elado` vagy `Vevo` mezőjének megfelelő mezőjébe kell beolvasni. Például az eladó nevét így is beolvashatjuk:

```
scanf("%s", Szerzodes.Elado.Nev)
```

A hajó adatait hasonló módon olvassuk be a `Hajo` változóba, majd ezt adjuk értékül a `Szerzodes` változó `Hajo` mezőjének. A vételárat a egyenesen a `Szerzodes Vetelar` mezőjébe olvassuk be, míg a szerződéskötés időpontja először a `Datum` segédváltozóba, majd onnan a `Szerzodes` struktúra `Keltezes` mezőjébe kerül.

A szerződés kiírásakor a `Szerzodes` struktúra megfelelő mezőit írjuk ki, a megfelelő szöveges körítéssel. Az eladó és vevő adatait először a `Kaloz` változóba töltjük, majd annak mezőit írjuk ki. Természetesen itt lehetne a `Szerzodes` struktúra `Elado` vagy `Vevo` mezőjének megfelelő mezőit egyből, segédváltozó használata nélkül is kiírni, pl.:

```
printf("\t\tNev: %s\n", Szerzodes.Elado.Nev);
```

A szerződés többi adatának kiírása hasonló módon történik.

Feladatok:

6.1. Készítsünk teknős-grafika segítségével olyan programot, amely kirajzol egy négyzetet a képernyőre, ahol a négyzet két átellenes pontjának koordinátái (10,10) és (90,90). A teknős-grafika elemi műveletei a következők:

- *Tollat letesz*: a teknős a papírra ereszti a toll hegyét, így ha a továbbiakban mozog, akkor vonalat húz maga után.
- *Tollat felvesz*: a teknős felemeli a tollat, így nem húz maga után vonalat akkor sem, ha mozog.
- *Odamegy*(*x,y*): A teknős az (x,y) koordinátájú pontba ballag a jelenlegi pozíciójából, mégpedig egy egyenes mentén haladva.
- *Előre megy*(*d*): a teknős a jelenlegi irányában halad előre d egységet.
- *Irányba áll*(*alfa*): A teknős az *alfa* irányba fordul (de nem halad). Az *alfa* szöget az *x* tengely pozitív irányához képest mérjük (0 fok a pozitív *x* irány, 90 fok a pozitív *y* irány, 180 fok a negatív *x* irány, stb.)
- *Elfordul*(*delta*): A teknős a jelenlegi irányához képest elfordul *delta* fokot.

(A programot írhatjuk pszeudo-kódban, vagy akár implementálhatjuk pl. Scratch alatt is: http://scratch.mit.edu/)

6.2. Készítsünk el a 6.1. feladat teknős-grafikája segítségével azt a programot, amely három darab négyzetet rajzol ki, amelyek bal alsó sarkának koordinátái rendre (10,10), (20,20) és (30,50). A négyzetek oldalai legyenek 20 egység hosszúak és párhuzamosak a koordináta-rendszer tengelyeivel.

Írjuk meg úgy a programot, hogy a lehető legkevesebb gondolkodással legyen módosítható a kód, ha máshová szeretnénk a négyzeteket helyezni. (Tipp: csak a négyzet első csúcsának elhelyezéséhez használjuk az *Odamegy* és *Irányba áll* parancsokat, a többi oldalt a relatív mozgásokat leíró *Elfordul* és *Előre megy* parancsokkal rajzoljuk meg.)

6.3. Rajzoljunk a 6.1. feladat teknős-grafikája segítségével egy szabályos hatszöget, melynek középpontja az origóban van, oldalai pedig 50 egység hosszúak. A hatszög két oldala legyen párhuzamos az *x* tengellyel.

Rajzoljunk hasonló módon 12-szöget. (Tipp: használjunk relatív mozgásokat leíró parancsokat.)

6.4. Morc Misi a szerződésekre ezentúl nem csak az eladó és a vevő, de az „ügyvéd” és a két tanú adatait rögzíteni akarja (természetesen ezek is kalózok).

Módosítsuk az adatszerkezetek tervét alkalmas módon.

Módosítsuk a program tervét alkalmas módon.

Módosítsuk a C nyelvű programot is.

6.5. A kalózok általában arany dukáttal fizetnek egymással kötött üzleteik során, így Morc Misi eredeti szerződés programját is csak erre készítették fel (a vételár arany dukátban volt értendő). Az utóbbi időben azonban felmerült az igény, hogy más fizetőeszközöket is lehessen a hajók adásvételénél használni (Féllábú Ferkó a múlt héten például egy tucat vak papagájért vett meg egy elsüllyedt kétárbocos szkúnert Szagos Szilárdtól).

Módosítsuk az adatszerkezetek tervét úgy, hogy a vételár megadható legyen tetszőleges fizetőeszközben is. Ehhez a vételár tartalmazza a fizetőeszköz típusát (pl. „arany dukát”, „rum (literben)”, „vak papagáj (darab)”, „rozsdás cipőkanál (darab)”, stb.), valamint a fizetőeszköz mennyiségét is (pl. 25, 12.5, 300).

Módosítsuk a program tervét is, hogy a különféle fizetőeszközöket is kezelje.

Módosítsuk a C nyelvű programot is. Ügyeljünk rá, hogy a kiírás elegánsan történjen.

6.6. Készítsünk olyan programot, ami a zsákmány adminisztrálását segíti Morc Misiéknek. A zsákmány tartalmaz aranypénzt, ezüstpénzt, ékszereket (ezek lehetnek arany- vagy ezüstékszerek, drágakövek, és egyebek), hajókat és túszokat (váltságdíj reményében). A programnak ezen elemek mennyiségét kell beolvasni és a végén tetszetős formában kiírni.

Készítsük el az adatszerkezetek tervét.

Készítsük el a program tervét.

Implementáljuk a programot C nyelven, struktúrák felhasználásával.

6.7. Egészítsük ki a 6.6. programot oly módon, hogy a zsákmány minden egyes tételéhez hozzá lehessen rendelni a felelős személyt, aki a zsákmány azon részét gondozza. A felelős személye mellett lehessen konkrét feladatot is megadni, amely a felelős tevékenységét definiálja: pl. a túszokért Galamb Gergő a felelős, feladata pedig a váltságdíj begyűjtése, míg a drágaköveket Enyves Ernőnek kell orgazdáknál értékesíteni. A felelős személyénél a szerződésben található adatokat tároljuk.

7. fejezet

Szelekciót tartalmazó adat- és programszerkezetek

Programjaink viselkedése gyakran a bemenő adatok függvényében változik. Ilyen feltételes viselkedési formák leírására a szelekciót tartalmazó adat- és programszerkezetek használhatók. Programjainkban nagyon gyakori lesz az elágazás, ami a szelekció megjelenése vezérlési szerkezetek szintjén. Ennél lényegesen ritkábban fogunk találkozni olyan adatszerkezetekkel, amik szelekciót tartalmaznak.

7.1. példa: Döntsük el három pozitív számról, hogy azok lehetnek-e egy háromszög oldalai.

Elemi geometriából ismert, hogy ha mindhárom számra igaz, hogy kisebb a másik két szám összegénél, akkor – és csak akkor –a három szám egy háromszög oldalhosszait adja meg. A feladatot a következő egyszerű C program oldja meg [*7.haromszoge.c*]:

```
/*
 * Eldönti, hogy három szakaszból szerkeszthető-e háromszög
 */
#include <stdio.h>

int main(){
   double a, b, c; /* a háromszög oldalai */
   printf("Adja meg az elso szakasz hosszat: ");
   scanf("%lf", &a);
   printf("Adja meg a masodik szakasz hosszat: ");
   scanf("%lf", &b);
   printf("Adja meg a harmadik szakasz hosszat: ");
   scanf("%lf", &c);
   /* Háromszög akkor szerkeszthető, ha bármely két szakasz
      hosszának összege nagyobb, mint a harmadik szakasz hossza. */
   if (a+b>c && a+c>b && b+c>a) {
      printf("A megadott szakaszokbol szerkesztheto haromszog.\n");
      printf("Gratulalok.\n");
```

```
}else
      printf("Sajnos nem szerkesztheto haromszog.\n");
   return 0;
}
```

A fenti program a C nyelvben használatos elágazás használatát mutatja be. Az egyágú elágazás a C nyelvben a következő formájú:

```
if (feltétel) utasítás
```

Az *utasítás* akkor hajtódik végre, ha a feltétel igaz. (A C nyelvben egy kifejezés hamis, ha annak értéke nulla, ellenkező esetben igaznak minősül). Pl. ilyen egyszerű feltételek lehetnek:

- `A>0` (A nagyobb nullánál),
- `A==B` (A egyenlő-e B-vel. Figyelem: logikai kifejezésekben két egyenlőségjel szükséges, a szimpla egyenlőségjel értékadáskor használatos.)
- `B!=5` (B nem egyenlő öttel)
- `A>=B` (A nagyobb, vagy egyenlő, mint B)
- `A<=3` (A kisebb, vagy egyenlő, mint 3)

Figyelem: gyakori hiba, hogy az egyenlőség vizsgálatakor a B=5 formát használjuk. Ilyen esetekben ugyan szintaktikailag helyes kódot írtunk, hiszen a B=5 egy kifejezés, így értéke is van (jelen esetben az értékadás eredménye, azaz 5 lesz a kifejezés értéke), ami nem nulla, tehát a feltétel igaz, függetlenül B korábbi értékétől. Ráadásul a B változó értékét felül is írtuk... Haladó C programozók az `if(A=B)` formát használják a következő kódszekvencia kifejezés rövid jelölésére:

```
A=B;
if(A)utasítás.
```

Az `utasítás` lehet egyetlen vagy több utasítás is, ez utóbbi esetben az utasításokat kapcsos zárójelek közé kell tenni (ezt a C nyelvben blokk, vagy összetett utasításnak nevezik), mint az a példában is látható.

A kétágú elágazások formája a C nyelvben a követező:

```
if(feltétel)
   utasítás1
else
   utasítás2.
```

Ha a *feltétel* igaz (nem nulla), akkor az `utasítás1`, ellenkező esetben pedig az `utasítás2` hajtódik végre.

Többágú elágazásokat értelemszerűen lehet készíteni kétágú elágazásokba ágyazott elágazások segítségével. Pl. a 3.4. ábra szerinti elágazás C nyelven a következő alakú lesz:

```
if(feltétel1)
   tevékenység1
else if(feltétel2)
   tevékenység2
else if(feltétel3)
   tevékenység3
else
   tevékenység4
```

Az egyszerű feltételekből logikai operátorokkal összetett logikai kifejezéseket is lehet készíteni. A C nyelvben az ÉS operátor jele a `&&`, a VAGY operátor jele a `||`, míg a logikai negálást a `!` jelöli. Néhány példa összetett logikai kifejezésekre:

- `A>0 || B <10` (A nagyobb nullánál, vagy B kisebb, mint 10),
- `A==1 && B!=3` (A egyenlő 1-gyel és B nem egyenlő 3-mal)
- `!(B>A)` (B nem nagyobb A-nál)
- `A==0 && B==2 || C>2` (A egyenlő nullával és B egyenlő kettővel, vagy pedig C nagyobb, mint 2)
- `(A==0 && B==2) || (C>2)` (Azonos az előzővel, zárójelezve)
- `(A==0 || B==2) && (C>2)` (A egyenlő nullával vagy B egyenlő kettővel, és ezen kívül C nagyobb 2-nél)
- `A==0 || B==2 && C>2` (A egyenlő nullával vagy pedig B egyenlő kettővel és C nagyobb, mint 2. Nem azonos az előzővel!)
- `A==0 || (B==2 && C>2)` (Azonos az előzővel, zárójelezve)
- `A && B==3` (az A nem nulla és a B hárommal egyenlő)
- `!A==2` (az A negáltja nem egyenlő kettővel)
- `!(A==2)` (nem igaz, hogy A egyenlő kettővel, nem azonos az előzővel!)

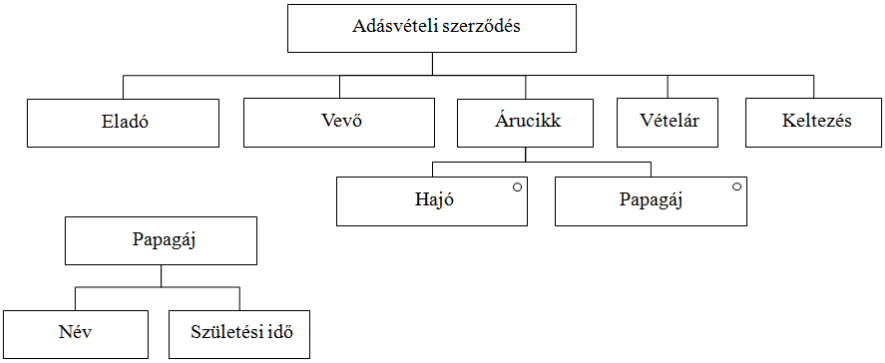
A legtöbb programozási nyelvben a kifejezésekben szereplő műveletek elvégzésének sorrendjét a nyelvben rögzített precedencia szabályok határozzák meg. A C nyelvi operátorok és precedenciáik listája az F3. Függelékben található, ami alapműveletek tekintetében a matematikában megszokott módon alakul. A legmagasabb precedenciája a zárójelnek van, tehát ezzel tetszés szerint csoportosíthatók az operációk. Ha nem vagyunk biztosak az operátorok precedenciájában, akkor zárójelezzünk nyugodtan. Egy programozási nyelven leírt kifejezés azonban nem csak a matematikai értelemben vett műveletet írja le, hanem a művelet operandusai által meghatározott értékek előállításáról, elővételének mikéntjéről is rendelkezik. Ezért a művelet elvégzése és az abban szereplő operandusok kiértékelési sorrendje időben elválhat. A legtöbb programozási nyelven az operandusok értékeinek előállítási sorrendje a fordító program magánügye. Ez akkor jelenthet problémát, ha az operandus értékének előállítása valamilyen mellékhatással visszahat valamelyik operandus értékére. Ilyen esetben a kifejezésünk értéke nem meghatározható a zárójelek tömegével sem. Ezt jól meg kell jegyeznünk különösen a C nyelv esetében, ahol számos olyan operátorunk van (++, --), aminek van mellékhatása is.

Példánkban az elágazás feltétele `a+b>c && a+c>b && b+c>a`, ami három egyszerű logikai kifejezés logikai ÉS kapcsolata. Az elágazás igaz ága a példa kedvéért két `printf()` utasítást tartalmaz, tehát ezeket kapcsos zárójelekkel egyetlen blokk-utasítássá fogtuk össze. Az elágazás különben ága csak egyetlen utasítást tartalmaz, ide nem kell a kapcsos zárójel. Ennek ellenére jó programozói gyakorlat a kapcsos zárójelek használata ilyen esetekben is, mert egyrészt jobban áttekinthetővé teszi a programot, másrészt a későbbi bővítés esetén nem felejtjük el a kapcsos zárójelet kitenni.

A kalózok a hajók adásvételénél pozitív tapasztalatokat szereztek: jelentősen csökkent a verekedésből adódó sérülések és halálesetek száma. Az előnyöket szeretnék kamatoztatni a másik jelentős kereskedelmi tevékenység, a papagájok adásvételének adminisztrációjára is. Morc Misi megbízta Sunyi Sanyit, hogy fejlessze tovább a programot.

7.2. példa: Segítsünk Sunyi Sanyinak kiegészíteni a már elkészült hajó-adásvételi programot oly módon, hogy az papagájok adásvételére is alkalmas legyen. A hajók adatait változatlan formában kezelje, míg a papagájok esetében a nevüket, valamint a születési dátumukat kell bekérni és kiírni.

A program nagyon hasonló lesz a hajók adásvételét kezelő programhoz, sőt annak nagy részét újra fogjuk hasznosítani. A szerződésben természetesen a hajó adatokon kívül egy újabb árucikk, a papagáj adatait is tudni kell tárolni. Ezen kívül a programban tudni kell beolvasni és kiírni mind a hajó, mind a papagáj adatait – egy szerződés esetén persze csak az egyiket a két lehetőség közül. A szerződés adatszerkezetét a 7.1. ábra mutatja: a 6.2. ábrához képest változás az, hogy a hajó helyett a szerződés egy árucikket tartalmaz, ami vagy hajó, vagy papagáj lehet. A papagájnak a nevét és születési idejét tároljuk. Az ábrán nem részletezett elemek definíciója megegyezik a 6.2. ábrán látható definíciókkal.



7.1. ábra. A hajók és papagájok adásvételét adminisztráló program adatszerkezetének leírása. Az ábrán nem részletezett elemek további definíciója a 6.2. ábrán található.

A program továbbra is beolvasásból és kiírásból áll, hasonlóan az eredeti program 6.3. ábrán látható felépítéséhez. Itt azonban a beolvasás és a kiírás kissé módosul: vagy hajót, vagy papagájt kell kezelni, ahogy az a 7.2. ábrán látható. A beolvasásnál először el kell dönteni, hogy melyik árutípust akarjuk beolvasni (hajót vagy papagájt): ehhez beolvassuk az áru típusát. Ezután vagy az egyik, vagy a másik típusú áru adatait olvassuk be. A kiírás során az árucikk típusának függvényében vagy az egyik, vagy a másik árutípus adatait írjuk ki.

A program C nyelvű kódjának részletei az alábbiakban láthatók. A tömörség kedvéért a korábbi adásvétel programmal megegyező részeket egyszerűen elhagyjuk, csak az új, illetve megváltozott részeket ismertetjük. A teljes program az elektronikus mellékletben található [*7.papagaj1.c*].

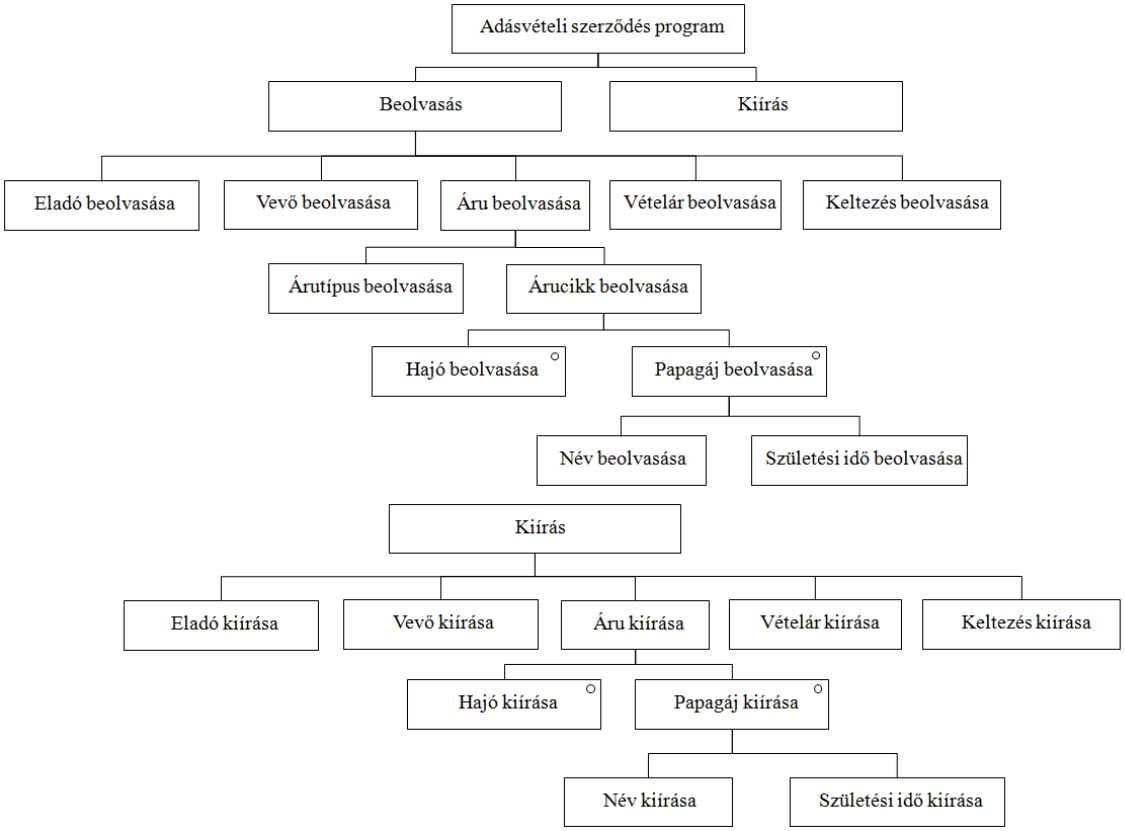
A papagájok adatainak tárolására definiáljuk a `papagaj` típusú struktúrát a `Nev` és `Szuletesi_ido` mezőkkel:

```
struct papagaj{
   char Nev[_MAX_HOSSZ+1];          /* a madár neve */
   struct datum Szuletesi_ido;      /* a madár születési ideje */
};
```

Az áruk típusát az `arutipus` típusú változókban fogjuk tárolni.

```
enum arutipus{HAJO, PAPAGAJ};
```

Az `enum` a C nyelvben olyan típusok definiálására szolgál, ahol a változók értékkészlete egy véges halmaz elemeiből áll. Jelen esetben az `enum arutipus{HAJO, PAPAGAJ}` utasítás egy olyan `arutipus` nevű típust definiál, amelyben a változók értéke vagy `HAJO`, vagy `PAPAGAJ` lehet (ezeket a fordító egy egész számmá konvertálja, tehát a `HAJO` helyett pl. nullát, a `PAPAGAJ` helyett pedig 1-et tárol, de ezzel nekünk nem kell törődnünk).



7.2. ábra. A hajók és papagájok adásvételét adminisztráló program leírása. Az ábrán nem részletezett tevékenységek definíciója meggyezik a 6.3. ábrán található definíciókkal.

Az árukat egy `aru` típusú struktúrában tároljuk.

```
struct aru{
  enum arutipus Arutipus;    /* az áru típusa */
  struct hajo Hajo;     /* ha az áru típusa hajó, akkor annak
adatai */
  struct papagaj Papagaj; /* ... ha papagáj, akkor annak adatai
*/
};
```

A 7.1. ábra szerint az áru vagy hajó, vagy papagáj lehet, de egyszerre csak az egyik. Az `aru` struktúratípus mezői nem egészen ezt a tervet tükrözik: a struktúrának van egy `Arutipus` nevű mezője, amely `arutipus` típusú (tehát értéke vagy `HAJO`, vagy `PAPAGAJ` lehet): az árutípus azonosítja, hogy a tárolt árucikk hajó-e vagy papagáj. Emellett a struktúra tartalmaz egy `Hajo` *és* (!) egy `Papagaj` nevű mezőt, amelyek ugyan mindketten jelen vannak a struktúrában, de csak az áru típusának megfelelőt fogjuk használni (hajó esetén nyilván a `Hajo`-t, papagáj esetén a `Papagaj`-t). A 7.1. ábra szerinti terv szerint az árucikk *vagy* hajó,

vagy papagáj (és csak az egyik), itt mégis mindkettő számára helyet foglaltunk (ráadásul az árutípust is el kellett tárolnunk), tehát a struktúrában a szelekció helyett egy szekvenciát valósítottunk meg. Ennek oka az, hogy a C nyelv (és általában a programozási nyelvek többsége) nem biztosít elegáns lehetőséget a szelekciók megvalósítására adatszerkezetek szintjén. Egy kicsit fejlettebb megoldást azonban erre a problémára is hamarosan adunk.
A `szerzodes` struktúratípusban megjelenik az `Arucikk` mező (a korábbi `Hajo` helyett):

```
struct szerzodes{
  struct kaloz Elado;        /* az eladó adatai */
  struct kaloz Vevo;         /* a vevő adatai */
  struct aru Arucikk;        /* az áru adatai */
  unsigned int Vetelar;      /* a hajó vételára */
  struct datum Keltezes;     /* a szerződés kelte */
};
```

Néhány új változót is használunk: létrehozunk egy `papagaj` típusú `Papagaj` nevű változót, amiben átmenetileg a papagájok adatait tároljuk (bekéréskor és kiíráskor), hasonlóan az `aru` típusú `Arucikk` nevű struktúrát az árucikk átmeneti tárolására fogjuk használni, míg az `arutipus_input` nevű kételemű karaktertömbben a felhasználó által begépelt, az áru típusát azonosító karaktert (hajó esetén h betűt, papagáj esetén p betűt) fogjuk tárolni:

```
struct papagaj Papagaj;
struct aru Arucikk;
char arutipus_input[2]; /* típus jelzése: (h)ajó vagy (p)apagáj*/
```

Az `arutipus_input` változó egy kételemű karakter tömb. Ezekkel részletesebben a 8. fejezetben fogunk foglalkozni. Most csak egy rövid magyarázat ennek használatáról programunkban:
A változót arra fogjuk használni, hogy egyetlen karaktert tároljunk benne. Ezt a karaktert egyelemű karakterláncként olvassuk be a `scanf()` függvény segítségével. A tömb első eleme tehát maga a beolvasott karakter lesz, a második eleme pedig a karakterláncot lezáró karakter (null-karakter): ezért szükséges a kételemű tömb. A tömb első eleme a C nyelvben a nulladik indexet viseli, tehát erre az `arutipus_input[0]` kifejezéssel hivatkozhatunk.

A program további részében az adatok bekérése és kiírása a korábbiak szerint zajlik, csak az áruk kezelése változott. Az árucikk bekérésekor először bekérjük az áru típusát az `arutipus_input` nevű változóba a `scanf` függvénnyel. Ha ez egy h betű, akkor egy hajó adatait kérjük be (a korábbiak szerint), ha ez egy p betű, akkor egy papagájét, más esetben pedig hibajelzést adunk.

```
printf("\n\tAru adatai:\n");
printf("\t\tAru tipusa:\n\t\t\t(h)ajo vagy (p)apagaj?");
scanf("%1s", arutipus_input);
if(arutipus_input[0]=='h'){
     /* hajó adatainak bekérése */
```

```
        ...
        Arucikk.Arutipus=HAJO;
        Arucikk.Hajo=Hajo;}
    else if(arutipus_input[0]=='p'){
        /* papagáj adatainak bekérése */
        printf("\t\tPapagaj adatai:\n");
        printf("\t\t\tNev: ");
        scanf("%s", Papagaj.Nev);
        printf("\t\t\tSzuletesi ido [e.h.n.]: ");
        scanf("%d.%d.%d.", &Datum.Evszam, &Datum.Honap, &Datum.Nap);
        Papagaj.Szuletesi_ido=Datum;
        Arucikk.Arutipus=PAPAGAJ;
        Arucikk.Papagaj=Papagaj;
    }
    else{
        printf("Ismeretlen arutipus: %s\n ", arutipus_input);
        return -1;
    }
    Szerzodes.Arucikk=Arucikk;
```

Az árutípus bekérésekor a `scanf()` függvénnyel egy 1 karakter hosszúságú karakterláncot olvasunk be. Ezt a `"%1s"` formátumvezérlővel érjük el (a formátumvezérlők leírását lásd az F2. függelékben). Ez az utasítás csak egyetlen karaktert fog eltárolni, függetlenül attól, hogy milyen hosszú karaktersorozatot gépelünk be.

Az elágazás feltétele az `arutipus_input[0]=='h'` kifejezés: itt a bemenetként megadott karaktert (`arutipus_input[0]`) hasonlítjuk össze a h karakterrel. A C nyelvben a karaktereket szimpla idézőjelek között kell megadni. A `'h'` egyébként a ha betű ASCII kódját jelenti, tehát az `arutipus_input[0]=='h'` kifejezés ekvivalens az `arutipus_input[0]==104` kifejezéssel, csak az előbbi sokkal olvashatóbb.

Figyelem: a C nyelvben a karakterláncot a dupla idézőjel, a karaktert (egy darab karaktert) a szimpla idézőjel jelöli. A `"h"` ugyan egyetlen karakterből álló karakterlánc, de nem azonos a `'h'` karakterrel: a karakterlánc végét mindig a végjel (null-karakter) zárja.

A fenti megoldásban egy egyszerű példát láthatunk a bemenő adatok hibakezelésére is: ha az árucikk típusa nem h vagy p, akkor a program hibaüzenetet ad és kilép. Ezt persze meg lehetne oldani más módon is úgy, hogy programunk felhasználóbarát módon kezelje a hibát (pl. kérje be újra az elrontott bemenetet) de ezzel majd később foglalkozunk. Most azt figyeljük meg, hogy programunk hiba esetén a `return -1` utasítással ér véget, jelezve ezzel, hogy a végrehajtás nem volt sikeres.

A szerződés egyes elemeinek kiírása a korábbi verziónak megfelelően történik, a különbség az árucikk kiírásánál van csupán. Itt is elágazást alkalmazunk az árucikket leíró struktúra `Arutipus` mezőjének függvényében: ha az áru típusa `HAJO`, akkor a hajó adatait, ellenkező esetben a papagáj adatait írjuk ki.

```
Arucikk=Szerzodes.Arucikk;
```

```
if(Arucikk.Arutipus==HAJO){
   /* hajó adatainak kiírása */
   ...
}
else{
   /* papagáj adatainak kiírása */
   printf("\n\tPapagaj adatai:\n");
   Papagaj=Arucikk.Papagaj;
   printf("\t\tNev: %s\n", Papagaj.Nev);
   Datum=Papagaj.Szuletesi_ido;
   printf("\t\tSzuletesi ido: %d.%d.%d.\n",
           Datum.Evszam, Datum.Honap, Datum.Nap);
   printf("\n\tPapagaj ");
}
```

Okos Ottokár, aki a kalózok informatikai rendszereit (vagyis a múlt évben zsákmányolt laptopot és nyomtatót) üzemelteti és felügyeli, megkérdezte a szerződéskezelő program készítőjét, Sunyi Sanyit, sátáni vigyorral a bajusza alatt:

– Ha a jövő héten szükséges lenne, hogy a programot pisztolyok adásvételéhez is módosítsuk, akkor az `aru` struktúratípus így nézne ki, ugye? – és Sunyi Sanyi orra alá dugott egy kissé viseltes papírfecnit a következő irománnyal:

```
struct aru{
     enum arutipus Arutipus;
     struct hajo Hajo;
     struct papagaj Papagaj;
     struct pisztoly Pisztoly;
};
```

– Igen, jól látod Okos Ottokár, ebből is látszik, hogy nomen est omen, tényleg okos vagy – hízelgett a program készítője, bajt szimatolva a levegőben.

– De persze a három mezőből (`Hajo`, `Papagaj`, `Pisztoly`) csak az egyiket használjuk, a másik kettő üres marad, ugye? – kérdezte Ottokár, s vigyora egyre szélesebbre húzódott.

– Hogyne, nagyon jól látod Okos Ottokár, csak az egyiket használjuk, mégpedig az áru típusától függően, amit az `Arutipus` mezőben el is tárolunk – bólogatott buzgón Sanyi, a program készítője, amely bólogatás egyre inkább reszketésre emlékeztetett.

– És ha majd húsz fajta árut is fog kezelni a programunk, akkor mind a húsz árunak ott lesz a helye, de csak mindig egyet használunk, ugyebár? – kérdezte vészjósló hangon Ottokár, s vigyora már füléig ért.

– Igen, mindig csak az egyiket használjuk, a többit nem, ahogy mondod, Okos Ottokár – rebegte Sanyi, s közben apró lépésekkel a kajüt ajtaja felé igyekezett.

– És ha majd százféle árut kezelünk a programmal, akkor száz árunak lesz lefoglalva a helye, de csak egyet használunk, kilencvenkilencet pedig elpazarolunk, nemde? – ordította Ottokár s vigyora már grimasszá torzult.

– Igen, de úgyis van elég memória – suttogta alig hallhatóan Sunyi Sanyi, hátát a bezárt ajtónak vetve.

– Jössz itt nekem a Jackson-ábráiddal, felrajzolod nekem a szelekciót az adatszerkezetbe, majd utána ki akarod szúrni a szemem egy szekvenciával, te nyavalyás szárazföldi patkány? Az én memóriámat akarod te pazarolni, mi? – tajtékzott Ottokár.

A beszélgetés itt félbeszakadt, majd fél óra múlva folytatták, miután a legénység nagy örömére Sunyi Sanyit áthúzták a hajó alatt.

– Nos, tudsz-e valami okosabb megoldás? – kérdezte negédesen Ottokár.

– Union – prüszkölte Sanyi.

– Juni micsoda? Beszélj értelmesen és ne vizezd össze a perzsa szőnyegemet – mordult fel Ottokár.

– Union – krákogta a program készítője. Ezzel fogon megoldani az árucikk tárolását és nem pazarlom majd a memóriádat. És jobban fogja tükrözni a Jackson-ábrán felrajzolt tervet is.

– Ez a beszéd! – veregette meg Sanyi vállát Ottokár. – De miért vagy ilyen vizes? Menj, törölközz meg, még megfázol itt nekem.

A probléma struktúrával történő megoldása valóban nem elegáns. A szelekciók adatszerkezetek szintjén történő szebb megvalósítását támogatja a C nyelvben a union.

Az union a C nyelvben a szelekció megvalósítására szolgáló adatszerkezet. A unionban, hasonlóan a struktúrához, mezőket definiálhatunk, amelyekre ugyanúgy hivatkozhatunk, mint a struktúrák esetében:

```
union u_tipus {                    struct s_tipus {
   int egesz;                         int egesz;
   float tort;                        float tort;
   char karakter;                     char karakter;
};                                 };
union u_tipus U_pelda;             struct s_tipus S_pelda;
```

A fenti példában az `u_tipus` típusú `U_pelda` nevű unionban tárolt adat vagy int, vagy float, vagy char típusú lehet, amelyekhez rendre az `egesz`, `tort` és `karakter` mezőnevek tartoznak. Az egész értékadás pl. a következő lehet: `U_pelda.egesz=168.` Ha törtszámot szeretnénk tárolni benne, akkor az `U_pelda.tort=3.1415`, míg a karakter tárolására a `U_pelda.karakter='W'` parancsot használhatjuk.

A union, a struktúrától eltérően nem foglalja le az összes mezőnek megfelelő helyet, az egyes mezők ugyanazon a memóriaterületen foglalnak helyet. A fenti példában az `S_pelda` struktúra mérete akkora, mint a struktúra mezőinek méretének összege, tehát pl. egy C fordító 4+4+1=9 bájtot foglal le az `S-pelda` változónak (hiszen általában az int, a float és a char típusok rendre 4, 4, és 1 bájtot foglalnak el). Ezzel ellentétben az `U_pelda` változó mérete csak akkora lesz, mint a legnagyobb mezőjének mérete, tehát esetünkben 4 bájt: ebből az int és float mezők lefoglalják mind a 4 bájtot, míg ha karaktert tárolunk a unionban, az csak 1 bájtot fog használni (a másik három lefoglalt bájt ilyenkor „kárba vész"). A struktúrák és unionok memóriafoglalását a 7.3. ábra szemlélteti egy három mezőt tartalmazó adatszerkezetben.

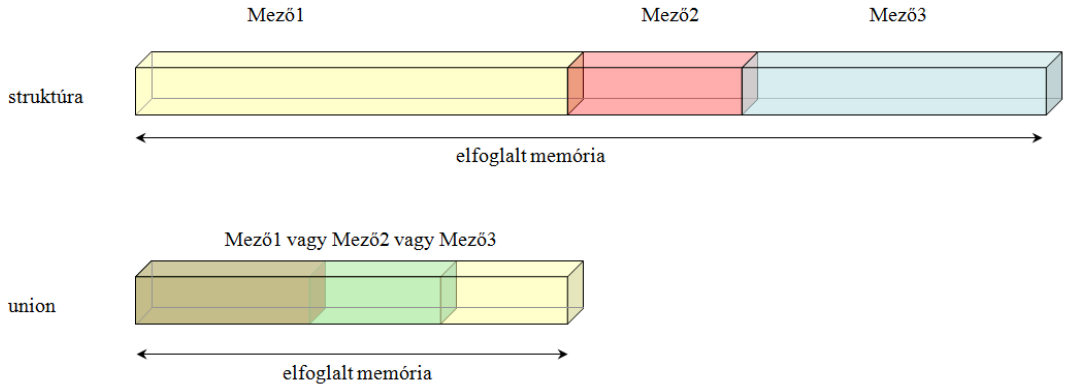
A uniont vagy az említett helytakarékossági okokból, vagy trükkös megoldások (pl. adatkonverziók) céljára szokás használni. Pl. a `union u{int szam; char bajtok[4];};` deklaráció lehetővé teszi, hogy az egészként beírt számot bájtonként olvassuk vissza.

Megjegyzés: mind a struktúrák, mind a union esetében a változók rögtön a típusdeklarációval együtt is létrehozhatók. Tehát a

```
union t_u{
      ...
};
union t_u u;
```

típus- és változódeklaráció ekvivalens a következővel:

```
union t_u{
      ...
} u;
```



7.3. ábra. Struktúrák és unionok elhelyezkedése a memóriában

7.3. példa: Segítsünk Sunyi Sanyinak a korábbi adásvételi programot úgy átírni, hogy az áruk adatait most union segítségével tárolja.

A union segítségével programunk adatszerkezete a következőképpen definiálható:

```
union aruadatok{              /*az áru adatainak tárolása*/
   struct hajo Hajo;          /* a hajó adatai */
   struct papagaj Papagaj;    /* a papagáj adatai */
};

struct aru{
   enum arutipus Arutipus;      /* az áru típusa */
   union aruadatok Aruadatok;   /* az áru adatai */
};
```

Az `aruadatok` típusú union két mezőt tartalmaz (`Hajo` és `Papagaj`), amelyek a már ismert `hajo` és `papagaj` típusú struktúrákat tartalmazzák – természetesen egyszerre csak az egyiket. Az `aru` struktúratípus az áru típusának azonosítására szolgáló `Arutipus` mezőn kívül tartalmaz még egy `aruadatok` típusú uniont az `Aruadatok` mezőben.

A hajó adatait tehát most a következőképpen lehet eltárolni:

```
Arucikk.Aruadatok.Hajo=Hajo;
```

A papagáj adatainak kiolvasása pedig a következőképpen történik:

```
Papagaj=Arucikk.Aruadatok.Papagaj;
```

A program egyéb részei megegyeznek a korábbi struktúrás megoldáséval. A teljes kód az elektronikus mellékletben megtalálható [*7.papagaj2.c*].

Megjegyzés: A union ugyan valóban hasznos eszköz lehet, de használata fokozott figyelmet igényel. Ezért csak olyan esetekben indokolt a használata, amikor a memória spórolására valóban igény van. A legtöbb személyi számítógépes alkalmazásban nincs jelentősége, hogy néhány bájtot vagy néhány kilobájtot foglalunk le (ezekre az esetekre érvényes Izmos Imi, a vén fedélzetmester örökérvényű mondása: „nincs értelme feleslegesen az árbocra mászni viharban biztosító kötél nélkül”). Ellenben egy kis beágyazott rendszerben, ahol pl. egy rádióüzenet mérete mindössze néhány tucat bájt, igencsak számít akár egyetlen elpazarolt bájt is. Ilyenkor a union, vagy akár a bitmezők alkalmazása is szükséges lehet (ez utóbbival nem foglalkozunk).

Feladatok:

7.1. Melyek az igaz logikai kifejezések a következők közül?

1
28
-2
3.14
0
1+1
3-4
1==3
-3+2+1
3>4
2>=2
!(1 || !(1 && 0))
!(0 || !0 && 1)
!(1 && !(1 || 0))

7.2. Mit ír ki a következő program a változók következő beállításainál?

- `a=3; b=4; c=-1; d=3;`
- `a=3; b=3; c=0; d=0;`
- `a=0; b=4; c=0; d=1;`
- `a=1; b=0; c=1; d=0;`

```
if ((a==b) || c && d)
       printf("%d\n", a+b-c);
else if (b || a && d)
       printf("%d\n", b+c-d);
else
```



```
        printf("%d\n", a+c-d);
```

7.3. Mit ír ki a következő program a változók következő beállításainál? (Vigyázat!)

- `a=3; b=2;`
- `a=3; b=3;`
- `a=1; b=5;`

```
printf("start - ");
if(a==b)
        printf("%d", a);
else if (b > 4)
        printf("%d", b-a);
else
        printf("%d", a+b);
        printf("%d", a+b);
printf(" – stop\n");
```

7.4. Módosítsuk az adásvétel adminisztráló programot úgy, hogy tudjon kezelni pisztolyokat is. A pisztolynak van típusa, kalibere és kalóz fegyver engedély száma (KFE szám).

7.5. Módosítsuk az adásvétel adminisztráló programot úgy, hogy tudjon kezelni fegyvereket. A fegyver lehet kés, pisztoly vagy borzkivonat. A késnek hossza van, a pisztolynak típusa és kalibere, a borzkivonatnak intenzitása és mennyisége (milliliterben). Valamennyi fegyvernek van ezen kívül kalóz fegyver engedély (KFE) száma.

7.5. Készítsünk számkitaláló játékot. A játékos gondol egy számot 1 és 10 között,a program pedig igyekszik minél kevesebb kérdésből ezt kitalálni.

7.6. Módosítsuk az adásvétel adminisztráló programot úgy, hogy tudjon kezelni fegyvereket. A fegyver lehet kés, pisztoly vagy borzkivonat. A késnek hossza van, a pisztolynak típusa és kalibere, a borzkivonatnak intenzitása és mennyisége (milliliterben). Valamennyi fegyvernek van ezen kívül kalóz fegyver engedély (KFE) száma.

7.7. Készítsünk programot, amely kiírja, hogy a megadott év szökőév-e.

7.8. Egészítsük ki a háromszög programot úgy, hogy el tudja dönteni egy háromszögről azt is, hogy egyenlő szárú vagy egyenlő oldalú-e, valamint azt is hogy derékszögű háromszög-e.

7.9. Készítsük el a másodfokú egyenlet megoldó programját. A program különböztesse meg, ha nincs, ha csak egy, illetve ha két valós gyök van.

7.10. Készítsünk programot, amely bekér egy dátumot (pl. a scanf("%d.%d.%d.", &Evszam, & Honap, &Nap) paranccsal), majd ellenőrzi, hogy a dátum legális dátum-e.

- A program ügyeljünk arra, hogy mely hónapokban hány nap lehet.
- Egészítsük ki a programot a szökőévek helyes kezelésével is.

7.11. A testtömeg indexet a testtömeg (kg) és a testmagasság (m) négyzetének hányadosából számítják (pl. 75/(1.78*1.78). Az alábbi táblázat tartalmazza a testtömeg index alapján való besorolásokat.

Testtömeg index	Besorolás
16 alatt	Kórosan sovány
16-20	Sovány
20-25	Normál testalkat
25-30	Túlsúlyos
30 fölött	Kórosan túlsúlyos

Készítsünk programot, ami bekéri a testtömeget és a testmagasságot, majd kiírja a testtömeg indexen alapuló besorolást.

8. fejezet

Iteratív adat- és programszerkezetek

Az iteratív adat és programszerkezetek rendkívül gyakoriak: ha egy tevékenységet többször kell elvégeznünk, iteratív programszerkezeteket használunk. Hasonlóan, ha egy adattípusból több áll rendelkezésre, akkor ezeket gyakran iteratív adatszerkezetekben tároljuk.

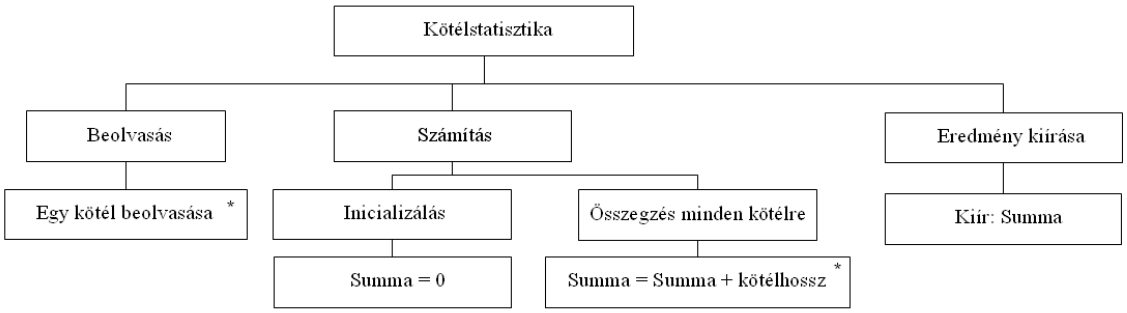
Kalóz Karcsi leltárt készít hajóján: szeretné tudni, hogy összesen hány méter kötél van a hajón, hány darabban, mennyi az átlagos kötélhossz, mennyi a legrövidebb és leghosszabb kötél hossza, és még számtalan statisztikai jellemzőt szeretne kiszámolni. Kötél mindenfelé előfordul a kalózhajón, a matrózok most méricskélik a kötéldarabokat. Kalóz Karcsinak egy segédprogramra lenne szüksége, ami beolvasná a mért adatokat és végül kiszámolná a statisztikai jellemzőket.

8.1. példa: Készítsünk egy programot Kalóz Karcsinak, ami beolvassa a kötelek adatait, eltárolja azokat, majd ebből kiszámítja a statisztikai jellemzőket. Egyelőre a kötelek összes hosszúságát számítsuk ki. A beolvasás során az utolsó kötél adatai után adjunk meg 0 hosszúságot, ezzel jelezzük a programnak, hogy az adatbevitel véget ért [*8.kotel1.c*].

A hajón sok kötél van, ezekről mind el kell tárolni a hosszát. Ezek ugyanolyan típusú adatok és sok van belőlük. Jackson-ábrás ábrázolással a 8.1. ábrán látható módon ábrázolhatjuk a kötelek adatait tároló adatszerkezetet.

8.1. ábra. A hajókötelek adatait tároló adatszerkezet

A programunk három nagy részből áll: beolvasás, amelynek során minden kötél adatát egyenként beolvassuk és eltároljuk, számolás, ahol a statisztikai jellemzőket (egyelőre a kötelek összes hosszát) kiszámítjuk, majd az eredmények kiírása következik. Mind a beolvasás, mind a számítás iteratív tevékenység lesz: a beolvasás során sokszor végezzük ugyanazt a tevékenységet: beolvassuk egy kötél adatait. A számítás során végigmegyünk az eltárolt adatokon és összeadjuk a kötelek eltárolt hosszát. Ezt a programszerkezetet illusztrálja a 8.2. ábra.



8.2. ábra. A hajókötelek összes hosszát számító program szerkezete

A C nyelvben az iteratív adatszerkezetek megvalósítására a tömbök szolgálnak. A tömb csak azonos típusú adatokból állhat. A tömb deklarációja a típus, a név, valamint a tömb méretének megadásából áll, pl.:

```
int it[33];     /* 33 elemű int típusú tömb*/
float ft[5];    /* 5 elemű float típusú tömb*/
char ct[100];   /* 100 elemű karaktertömb*/
```

Az `it` tömb 33 egész számot tartalmaz. Az egyes elemekre indexeikkel hivatkozhatunk:

```
it[4] = 5; x = it[21];
```

A C nyelvben a tömbök indexelése 0-val kezdődik, Vagyis egy N-elemű tömb első eleme a 0, a második eleme az 1, az utolsó eleme pedig az N-1 indexet kapja. Példánkban tehát az `it` tömb elemeit 0 és 32 közötti indexekkel lehet elérni, a `ct` tömb pedig 0 és 99 közötti indexekkel címezhető (a határokat is beleértve).

A fenti egyszerű adattípusokon kívül a tömbök elemei lehetnek összetett adatszerkezetek is (pl. tömbök, struktúrák), ezekkel majd a 10. fejezetben foglalkozunk.

A kötelek adatainak tárolására egy tömböt fogunk használni. A C nyelvben a tömb méretét előre meg kell adni (a dinamikus memóriakezelés lehetőségeivel itt most nem foglalkozunk.) Ezért Kalóz Karcsinak előre meg kell mondania, hány kötélről akar statisztikát készíteni. Persze nem kell pontosan tudnia a kötelek számát, de egy felső becslést kell mondani: pl. biztosan nem lesz 200 kötélnél több. Ekkor programunkban lefoglalunk egy 200 elemű tömböt és legfeljebb nem használjuk ki az egészet. Programunk szerkezete tehát így fog alakulni:

```
#include <stdio.h>
#define ELEMSZAM 200

float kotel[ELEMSZAM];
float summa;    /* ebbe kerül a végeredmény*/
float hossz;    /* az aktuális kötél hossza*/
int ix;         /* a tömb indexelésére használjuk */

int main(){
     /* beolvasás*/
     /* számítás*/
     /* kiírás*/
return 0;
}
```

Az `ELEMSZAM` makró definiálja a kötelek maximális számát (200), ennek segítségével adjuk meg a `kotel` nevű `float` típusú tömb méretét is. Az adatok beolvasásához és feldolgozásához szükségünk lesz a C nyelv ciklusszervező utasításaira is.

A C nyelvben a következő ciklusszervező utasítások találhatók:

- Elöltesztelő, bennmaradási feltétellel. Ennek szintaxisa a következő.

  ```
  while (feltétel) utasítás
  ```

  Természetesen az *utasítás* lehet kapcsos zárójelek között megadott blokk utasítás is. A ciklus végrehajtása során először a feltétel értékelődik ki: ha igaz, akkor végrehajtjuk a ciklusmagot (és kezdjük elölről a feltétel kiértékelésénél), különben kilépünk a ciklusból. A következő példa a while ciklus használatát mutatja be:

  ```
  c='a';
  while (c < 'g') {
      printf("%c ", c);
      c++;
  }
  printf("vege: %c ", c);
  ```

  A példabeli programban a ciklus előtt a `c` (karakter típusú) változónak kezdeti értéket adunk: az *a* karaktert. A ciklus bennmaradási feltétele, hogy az `c` változó értéke kisebb legyen a g betűnél. Mivel a karaktereket az ASCII kódjukkal reprezentáljuk (lásd az F1 függeléket), két betű kisebb-nagyobb relációja ez alapján dönthető el. Mivel az ASCII táblában a kódokat betűrendben rendelték a karakterekhez, így egy karakter akkor kisebb a g betűnél, ha előtte van a betűrendben (ez a kis betűkre igaz, de pl. az összes nagy betű kódja már kisebb, mit a kis betűké, lásd az ASCII táblát).

  A kódban látható `c++` utasítás a `c=c+1` utasítás rövid formája. A ++ operátorhoz hasonlóan létezik még a `--` operátor is: a `c--` kifejezés a `c=c-1` rövidítése. [1]

  A ciklus magjában tehát egyesével léptetjük a karaktereket (a, b, c, d, stb.), amíg a c változó értéke 'g' nem lesz; ekkor kilépünk a ciklusból. A futási eredmény tehát a következő lesz:

  ```
  a b c d e f vege: g
  ```

- Hátultesztelő, bennmaradási feltétellel. Ennek szintaxisa a következő.

  ```
  do utasítás while (feltétel);
  ```

  Az *utasítás* itt is lehet blokk utasítás is. Ennél a ciklusnál először végrehajtódik a ciklusmag, majd kiértékelődik a feltétel. Amennyiben a feltétel igaz, újra kezdődik a ciklus a ciklusmag végrehajtásával, ellenkező esetben kilépünk a ciklusból. Használatára egy példa:

  ```
  do {
      scanf("%d", &s);
      printf("%d negyzete: %d\n", s, s*s);
  } while (s >= 0);
  ```

[1] Az ilyen – úgynevezett mellékhatásos – operátorok kezelése körültekintést igényel. A `c=1; b=c++;` szekvenciában például a `c++` (poszt-inkremens) utasítás fő hatása, hogy a kifejezés értéke a `b` változóba töltődik (b=1), majd mellékhatásként `c` értéke inkrementálódik (c=2). Ezzel ellentétben a `++c` pre-inkremens operátor használatánál először a mellékhatás következik be (`c` inkrementálódik), majd a kifejezés kiértékelődik (fő hatás), így a `c=1; b=++c;` szekvencia után b=2 és c=2. Sokszor csupán mellékhatásukért használjuk ezen operátorokat, ilyenkor természetesen hatásuk azonos. Gyakran használjuk még a poszt-dekremens (`c--`) és pre-dekremens (`--c`) operátorokat is. Lásd még az F3. függeléket.

A fenti programrészlet a ciklus magjában beolvas egy egész számot majd kiírja a négyzetét. Teszi ezt mindaddig, amíg a beolvasott szám nem negatív. Itt a ciklus magja mindig végrehajtódik legalább egyszer, hiszen a feltétel ellenőrzése csak a ciklus végén történik meg.

- Számlálóvezérelt.

  `for(`*`inicializálás`*`; `*`feltétel`*`; `*`léptetés`*`) `*`utasítás`*

  Ez a ciklusforma egy speciális elöltesztelő ciklus, amelynek feltétele a ciklus fejében megadott *`feltétel`* kifejezés. Ennél a ciklusnál a ciklus végrehajtása előtt még végrehajtódik az *`inicializálás`* kifejezés, illetve a ciklusmag (az *`utasítás`*) minden végrehajtása után végrehajtódik a *`léptetés`* is. A ciklusmag itt (*`utasítás`*) is lehet blokk utasítás. Tipikus használatára példa:

```
for (i = 0; i < 10; i++) {
    printf("%d ", i);
}
printf("vege: %d ", i);
```

  Itt először az `i` változó kezdeti értéke állítódik be nullára (inicializálás), majd a ciklusmag hajtódik végre, ha az `i` változó értéke kisebb, mint 10 (feltétel). A ciklusmag végrehajtása után végrehajtódik a léptetés, ami jelen esetben az `i++` kifejezés: ez az `i` értékét eggyel megnöveli. Majd ismét a feltétel ellenőrzése, ciklusmag végrehajtása és a léptetés, stb. következik mindaddig, amíg a feltétel igaz. Amint a feltétel hamissá válik, kilépünk a ciklusból. Jelen példánkban az `i` értéke nulláról indul és egyesével nő a ciklusmag minden végrehajtása után. A tizedik végrehajtás után i értéke 10 lesz, vagyis a feltétel hamissá válik. Ekkor kilépünk a ciklusból. Programunk tehát a következő kimenetet produkálja:
`0 1 2 3 4 5 6 7 8 9 vege: 10`

Mivel legalább egy adatot be kell olvasnunk (ami az első kötél hossza, vagy ha egyetlen kötél sincs, akkor a záró nulla), ezért a beolvasáshoz használjunk először hátultesztelő ciklust:

```
/* maximum ELEMSZAM számú hossz beolvasása, míg hossz > 0 */
ix=0;      /* az aktuális tömbindex */
do {
     printf("adja meg a kovetkezo kotel hosszat: ");
     scanf("%f", &hossz);
     kotel[ix]=hossz;   /* tárolás a tömbben */
     ix++;              /* tömbindex növelése*/
} while(hossz>0 && ix<ELEMSZAM);
```

A ciklusmagban beolvassuk az aktuális kötél hosszát a `hossz` változóba, majd ezt eltároljuk a `kotel` tömb `ix` indexű elemébe. Ezután az indexet növeljük. A ciklus akkor folytatódik (bennmaradási feltétel!), ha a legutóbb beolvasott hossz nagyobb, mint 0 és még nem léptük túl a maximális elemszámot.

Figyelem:

- A beolvasó rutin eltárolja a tömbbe az utoljára beolvasott záró nulla értéket is. Ezt felhasználhatjuk a számítás során arra, hogy a tömbben megtaláljuk az utolsó érvényes adatot.
- Amennyiben a kötelek száma eléri a maximális (`ELEMSZAM`) értéket, az utolsó eltárolt adat nem a záró nulla érték lesz. Erre ügyelnünk kell majd a számítás végzésekor.

A számítás során végiglépkedünk a tömb elemein, amíg a záró elemig, vagy a tömb végéig el nem jutunk. A ciklusmagban az egyes eltárolt kötélhosszakkal növeljük a `summa` változó értékét, ami a ciklus végén a teljes kötélhosszt tartalmazza. Itt célszerűen egy elöltesztelő ciklust használhatunk:

```
/* számítás*/
ix=0;           /* az aktuális tömbindex */
summa=0;        /* részeredmények tárolója */
while (kotel[ix]!=0 && ix<ELEMSZAM){
     summa += kotel[ix]; /* aktuális kötélhossz hozzáadása*/
     ix++;    /* index növelése */
}
```

A ciklus feltételében a `kotel[ix]!=0` kifejezés csak a záró 0 értékig engedi futni a ciklust, míg az `ix<ELEMSZAM` azt az esetet kezeli, amikor a teljes tömb tele van érvényes adattal és nincs záró nulla. A ciklus addig fut, amíg mindkét feltétel igaz: nem értünk el záró nullát és nem értünk a tömb végére sem.

A programban használt `summa+=kotel[ix]` kifejezés a `summa=summa+kotel[ix]` rövidebb írásmódja. A C nyelvben még számos ilyen tömör értékadó operátor létezik:

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

Az operátorok jelentése az F3. függelékben található.

A kiírás során egyszerűen a `summa` változó értékét írjuk ki:

```
/* kiírás*/
printf("a kotelek teljes hossza: %f\n", summa);
```

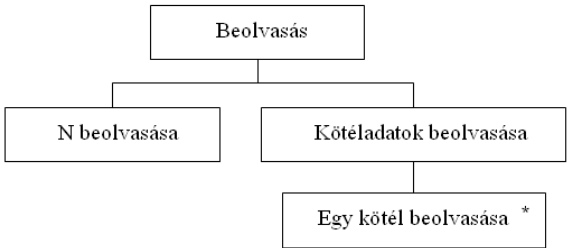
Kalóz Karcsi módosította a követelményt: először megszámolják, hogy hány darab kötél van a hajón és ezt az adatot bekéri a program. Utána a program egyenként bekéri minden kötél hosszát (nem szükséges az adatbevitel végét jelezni).

8.2. példa: készítsük el Kalóz Karcsi módosított kötélnyilvántartó programját [*8.kotel2.c*].

A módosított programban szükségünk lesz még egy változóra, amelyben a kötelek számát tároljuk, jelöljük ezt N-el:

```
int N;          /* a kötelek száma*/
```

A beolvasás során először a kötelek számát kell beolvasni, majd egyenként a kötelek adatait, ahogy a 8.3. ábra mutatja.



8.3. ábra. A módosított hajókötél-beolvasó felépítés

Mivel a beolvasandó kötelek száma ismert, itt célszerűen számlálóvezérelt ciklust alkalmazhatunk:

```
/* beolvasás*/
/* - kötelek számának beolvasása*/
printf("Hany kotel van? ");
scanf("%d", &N);

/* - N db kötél adatainak beolvasása*/
for(ix=0; ix<N; ix++){
     printf("adja meg a kovetkezo kotel hosszat: ");
     scanf("%f", &hossz);
     kotel[ix]=hossz;
}
```

A számítás során is kihasználjuk, hogy ismert a kötelek száma, itt szintén számlálóvezérelt ciklust alkalmazunk:

```
/* számítás*/
summa=0;    /* kötelek teljes hossza */
for(ix=0; ix<N; ix++){
     summa += kotel[ix];
}
```

A számlálóvezérelt ciklusok ilyen esetekben tömörebb, jobban áttekinthető kódokat eredményeznek, hiszen itt a ciklusváltozó inicializálását, növelését és a feltétel ellenőrzését egy helyen kezeljük a kódban. Ezt a tömör jelölési módot gyakran kihasználják a C nyelvben úgy, hogy a for-ciklust általános elöltesztelő ciklusok jelölésére használják.

Kalóz Karcsi a kötelek hosszán kívül szeretné tudni azt is, hogy mennyi a leghosszabb kötél hossza.

8.3. példa: Írjuk át a kötélnyilvántartó programot úgy, hogy a bekért adatokból a leghosszabb kötél hosszát is meghatározza [*8.kotelmax.c*].

A legnagyobb elem kereséséhez a bekért tömböt fogjuk használni. A keresés során végigmegyünk a tömb elemein egy ix tömbindex segítségével és az eddig talált legnagyobb elem indexét a maxix változóban tároljuk. A legnagyobb elem indexe kezdetben a tömb első elemének indexe lesz, ami C-ben 0, a tömb bejárását pedig a második (1 indexű) elemtől kezdjük el. A ciklusban összehasonlítjuk, hogy az aktuálisan vizsgált elem nagyobb-e, mint az eddigi

legnagyobb érték: ha igen, akkor módosítjuk a legnagyobb elem indexét. (A tömb bejárását kezdhetnénk a 0. indexű elemnél is, de így a ciklust egyszer feleslegesen hajtanánk végre, hiszen az tömb első elemét önmagával hasonlítanánk össze. A program természetesen helyesen működne így is.) A C nyelvű kód számlálóvezérelt ciklussal a következő lehet:

```
int maxix;  /* a legnagyobb elem indexét tároljuk itt*/

/* maximum számítás*/
maxix=0;     /* az eddigi legnagyobb elem indexe */
for (ix=1; ix<N; ix++){
     if (kotel[ix] > kotel[maxix])  /* ha az uj elem nagyobb... */
          maxix=ix;       /* ... akkor tároljuk el annak indexét*/
}

/* leghosszabb kötél kiírása */
printf("a leghosszabb kotel hossza: %f\n", kotel[maxix]);
```

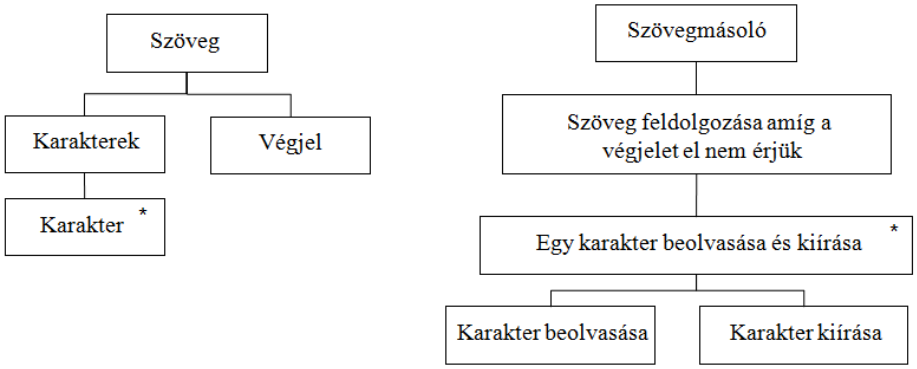
Figyeljük meg, hogy ennél a megoldásnál valójában nem a legnagyobb kötélhosszt kerestük meg, hanem a leghosszabb kötél indexét (`maxix`), a leghosszabb kötél hosszát nem tároltuk el. Ezért a kiírásnál is a tömb megfelelő indexelésével (`kotel[maxix]`) értük el a keresett hosszt.

Amennyiben az eredeti megoldást választjuk, ahol is a tárolt elemek száma nem ismert, de egy végjel mutatja a tömbben az érvényes adatok végét, akkor is egy hasonló elvű megoldást alkalmazhatunk, de a ciklus (pl. elöltesztelő) különböző lesz. A tömb bejárása azonos módon történik, mint azt az összegző programnál láttuk, de itt most csak az 1. indextől indulunk. Elöltesztelő ciklussal pl. így oldható meg a feladat:

```
/* maximum számítás*/
ix=1; maxix=0;
while (kotel[ix]!=0 && ix<ELEMSZAM){
     if (kotel[ix] > kotel[maxix])
          maxix=ix;
     ix++;
}
```

8.4. példa: Készítsünk programot, amely a bemenetén érkező karaktersorozatot karakterenként a kimenetre másolja [*8.karaktermasol1.c*].

A szöveg egy karaktersorozat, amelynek a végét egy speciális jel (EOF – end of file) jelzi, ahogy a 8.4. ábra bal oldalán látható. A feldolgozó program felépítése a 8.4. ábra jobb oldalán látható: a végjel eléréséig olvasunk be karaktereket és ezeket mindjárt ki is írjuk.



8.4. ábra. A szöveg felépítése és a karaktermásoló program szerkezete

A C nyelvben egyetlen karakter beolvasása a `getchar()` függvénnyel történik:

```
karakter=getchar();
```

Figyelem: a `getchar` függvény visszatérési értéke `int` típusú, és nem `char`. Jóllehet az ASCII karakterek elférnek a C nyelvben használt 1 bájtos `char` típusban, de a `getchar` függvény nem csak ezeket, hanem az egy bájtnál nagyobb méreten kódolt karaktereket (pl. 'ő') is helyesen próbálja kezelni, ezért szükséges a nagyobb méret.

Egy karakter kiírása a `putchar()` függvénnyel lehetséges:

```
putchar(karakter);
```

Mivel programunkban legalább egy karaktert be kell olvasni, próbáljunk meg egy hátultesztelő ciklust használni:

```
/*
 * karaktermásoló a standard bemenetről a standard kimenetre
 */

#include <stdio.h>
main(){
int ch;  /* a karakter átmeneti tárolója */
   do {
      ch=getchar();       /* beolvasás a standard bemenetről */
      putchar(ch);        /* kiírás a standard kimenetre */
   } while (ch != EOF);   /* a fájl végéig */
}
```

Az fájlok végét valójában nem zárja le EOF karakter, fizikailag nem tárolunk ilyen karaktert a fájlok végén. A fájlok végét az operációs rendszer észleli és ezt jelzendő adja vissza a speciális EOF karaktert programunknak.

Amikor fájl helyett billentyűzetről olvasunk, akkor viszont nekünk kell jeleznünk a bemenet végét, az utasítás számára a „fájl végét". Ezt Linux alapú rendszereken a CTR-D karakter megnyomásával tehetjük meg, míg Windows alatt a CTR-Z karakter és az ENTER megnyomása generálja az EOF karaktert.

A fenti megoldás működik ugyan, de nem szép: programunk az EOF karaktert is megpróbálja kiírni a többi karakterhez hasonlóan, ami értelmetlen. Szebb megoldás, ha ezt nem teszszük: egy elágazás segítségével akadályozzuk meg az EOF kiírását. Ekkor a ciklus a következőképpen néz ki:

```
do {
   ch=getchar();
   if (ch != EOF) putchar(ch);
} while (ch != EOF);
```

8.5. példa: Valósítsuk meg karaktermásoló programunkat elöltesztelő ciklussal [*8.karaktermasol2.c*].

Mivel itt a ciklus elején történik a annak tesztelése, hogy elértük-e már a fájl végét, így a ciklus előtt szükséges az első karakter beolvasása:

```
#include <stdio.h>
int ch;
main()
{
   ch=getchar();       /* ciklus előtt az első adat beolvasása
*/
   while (ch != EOF)  /* ismétlés, míg a fájl végét el nem
érjük*/
   {
      putchar(ch);     /* az előbb beolvasott karakter kiírása*/
      ch=getchar();    /* következő karakter beolvasása*/
   }
}
```

Míg a hátultesztelő ciklus magjában először beolvassuk a karaktert majd kiírjuk azt, az elöltesztelő ciklusnál a ciklusmagban most először a korábban (előző ciklusban, vagy a ciklus előtt) beolvasott karaktert írjuk ki, majd beolvassuk a következő ciklusban kiírandó karaktert. A következő ciklusba természetesen csak akkor lépünk be, ha nem fájlvégjelet olvastunk, így az EOF kiírásával nem próbálkozunk meg.

A fenti ciklust a C nyelv lehetőségeit kihasználva még tömörebben is le lehet írni:

```
while ((ch=getchar()) != EOF)
   putchar(ch);
```

A C nyelvben az értékadó is kifejezésnek is van értéke: a kifejezés értéke maga a jobb oldal értékével egyezik meg. Tehát az `x = 1` kifejezés értéke 1. Példánkban a `ch = getchar()` kifejezés értéke a `getchar()` által visszaadott érték (a beolvasott karakter) lesz.

Programunk ciklusának feltétel részében tehát beolvassuk a karaktert, ezt értékül adjuk a ch változónak, majd megvizsgáljuk, hogy a beolvasott érték nem EOF-e.

Vigyázat: az operátorok precedenciája miatt az egyenlőtlenség vizsgálat hamarabb kiértékelődne, mint az értékadás, ezért a következő kód rossz eredményt ad:

```
while (ch = getchar() != EOF)    /* ROSSZ */
```

A `ch = getchar()` kifejezés zárójelezése szükséges:

```
while ((ch = getchar()) != EOF)  /* JÓ */
```

Az értékadó kifejezések láncolásával a C nyelvben pl. lehetséges a következő értékadás: `x=y=1`. Itt először az `y=1` értékadás történik meg, amely kifejezés értéke (1) adódik értékül az x változónak.

A tömör, trükkös megoldások használata természetesen nem feltétlenül szükséges ahhoz, hogy jól működő, szép kódot írjunk. Megértésük azonban szükséges lehet, ha mások által írott kódot olvasunk.

Ha programunkat a masol.c forrásból a masol.exe állományba fordítjuk, akkor a futtatás pl. így történhet:

```
masol.exe < masol.c
```

Ekkor a program a masol.c fájl tartalmát kapja bemenetül és így kiírja a képernyőre forrásprogramunk tartalmát. (Erről az operációs rendszer gondoskodik, amelyet a < karakterrel utasítottunk, hogy a masol.c nevű fájl tartalmát irányítsa át a standard bemenetre.)

Ha a programot egyszerűen a masol.exe paranccsal indítjuk, akkor a program a billentyűzetről várja a bemenetet. Ilyenkor gépelhetünk szöveget és a bevitel végét a CTRL-D (Linux) vagy CTRL-Z és ENTER karakterekkel (Windows) jelezzük.

Figyelem: Ha billentyűzetről adjuk meg a bemenő szöveget, akkor a bemenet pufferelése miatt csak az ENTER leütése után történik meg a feldolgozás: ilyenkor programunk a begépelt szöveget soronként ismétli meg.

8.6. példa: Az 5.7. ábra egy karakterszámoló program felépítését mutatja, amely egy bemenetként megadott szövegben megszámolja a kisbetűket, nagybetűket és egyéb karaktereket. Implementáljuk a programot C nyelven [*8.karakterszamol.c*].

A program struktúrája megegyezik az 5.7. Jackson-ábrán láthatóval:

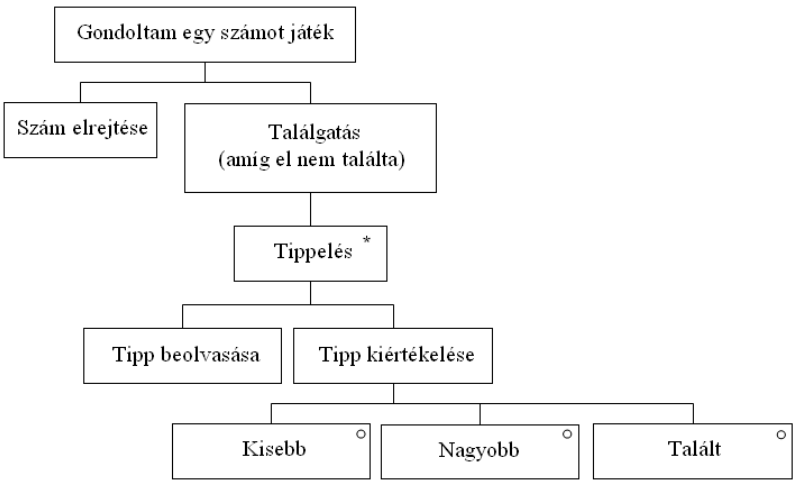
```
#include <stdio.h>
int ch;
int K;   /* kisbetűk számlálója */
int N;   /* nagybetűk számlálója */
int E;   /* egyéb karakterek számlálója */
main()
{
K=N=E=0;              /* inicializálás */
   ch=getchar();      /* első karakter beolvasása */
   while (ch != EOF)
   {
      if('A'<=ch && ch <='Z'){
         N++;   /* nagybetűk számolása */
      } else if ('a'<=ch && ch <='z'){
         K++;   /* kisbetűk számolása */
```

```
            } else {
                E++;   /* egyéb karakterek számolása */
            }
            ch=getchar(); /* következő karakter beolvasása */
        }
        /* eredmények kiírása */
        printf("A kis betuk szama: %d\n", K);
        printf("A nagy betuk szama: %d\n", N);
        printf("Az egyeb karakterek szama: %d\n", E);
    }
```

Kalóz Karcsi rákapott a számítógépes játékokra. Hogy a rumivástól megfogyatkozott agysejtjeit is edzeni tudja, egy logikai játékprogramot készíttet. A program gondol egy számot, amit Karcsinak minél kevesebb számú tippből ki kell találni. A program minden tippnél elárulja, hogy a gondolt szám kisebb-e vagy nagyobb, mint a tipp. Ha a tipp egyenlő a gondolt számmal, akkor a program meleg gratulációval véget ér.

8.7. példa: Készítsük el Kalóz Karcsi számkitaláló játékát [*8.szamkitalal.c*]

A program először elrejt egy számot („gondol”), majd a felhasználó addig tippel, amíg el nem találja a gondolt számot. A program minden tippet kiértékel, ahol a kiértékelés eredménye a kisebb, nagyobb, vagy egyenlő (talált) lehet. A program felépítése a 8.5. ábrán látható.



8.5. ábra. A Gondoltam egy számot játék szerkezete

Mivel a ciklusban legalább egy tippet fel kell dolgozni, jó választás a hátultesztelő ciklus. A ciklus magjában egy elágazás segítségével értékeljük ki a tippet. A program C nyelvű implementációja a következő lehet:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define MAX 100

int main(){
      int szam;          /* a gondolt szám */
      int tipp;          /* a tippelt szám */
      srand(time(NULL)); /* a véletlenszám-generátor inicializálása */
      szam=rand()%MAX+1; /* véletlenszám 1 és MAX között */
      printf("Gondoltam egy szamot 1 es %d kozott, talald ki!\n",
             MAX);
      do {
            printf("Tipped? ");
            scanf("%d", &tipp); /* tipp bekérése */
            /* tipp kiértékelése */
            if(tipp<szam){
                  printf("A szam nagyobb. ");
            }
            else if(tipp>szam){
                  printf("A szam kisebb. ");
            }
            else{
                  printf("Kitalaltad, gratulalok!\n");
            }
      } while(tipp!=szam);   /* ismétlés, míg ki nem találjuk... */
      return 0;
}
```

Véletlen számok generálására a `rand` függvény használatos, amely 0 és egy nagy érték (általában 32765) közötti egész számot ad vissza. Programunkban a `szam=rand()%MAX+1` utasítás maradékos osztás segítségével egy 0 és MAX+1 közötti tartományra transzformálja a véletlen számot. Mivel a `rand` függvény valójában egy álvéletlen generátorral működik, így a generált véletlen szám a program minden futtatásakor ugyanaz lenne. Ezért a generátor kezdeti értékét általában az aktuális idő függvényében állítjuk be az `srand` függvénnyel, a programban pl. a `srand(time(NULL))` parancsot használtuk erre a célra. A `time` függvény az 1970 január 1, 0:00 óra óta eltelt másodpercek számát adja vissza.

Feladatok:

8.1. Írjuk át a karakterszámláló programot (5.7. ábra) hátultesztelő ciklus használatával.

8.2. Írjuk át a karakterszámláló programot, hogy a számokat is külön számolja meg, ezen kívül az egyéb karakterek között csak a látható karaktereket számolja. A látható karakterek a felkiáltójel (0x21) és a hullámvonal (0x7E) között vannak (lásd az ASCII táblát az F1. függelékben).

8.3. Írjunk programot, ami kiírja a négyzetszámokat 1-től 1000-ig.

8.4. Írjunk programot, ami bekér egy számot, majd kiírja a négyzetszámokat 1-től a bekért szám négyzetéig.

8.5. Írjunk programot, ami táblázatot készít a Celsius-Fahrenheit értékpárokról. A táblázatban elöl álljon a Celsius érték (-40 foktól +40 fokig terjedő intervallumban, egyesével), majd mögötte az ekvivalens Fahrenheit–érték álljon. Fahrenheit fokba a következő képlettel lehet átváltani a Celsiusban megadott értéket: $X_F = 5/9(X_C - 32)$.

8.6. Írjuk át a Gondoltam egy számot programot úgy, hogy elöltesztelő ciklust használjon.

8.7. Írjuk át a Gondoltam egy számot programot úgy, hogy számolja meg a tippelések számát és ennek függvényében gratuláljon.

8.8. Írjuk át a Gondoltam egy számot programot úgy, hogy számolja meg a logikus és nem logikus tippek számát és ezek függvényében gratuláljon. Logikus a tipp, ha az eddigi ismert minimum és maximum között van, ellenkező esetben nem logikus.

8.9. Készítsünk egy hisztogramrajzoló programot. A program generál 100 darab véletlen számot az 1-10 intervallumban, majd ezen véletlen számok előfordulási gyakoriságát megjeleníti egy fekvő hisztogram formájában. A függőleges tengelyen az előforduló értékek, míg a vízszintes tengelyen ezek gyakorisága szerepel. A gyakoriságot annyi csillag karakter mutatja, ahányszor az adott érték előfordult. Ügyeljünk arra, hogy a gyakoriság tengely a legnagyobb gyakoriságig legyen beszámozva. A program futási eredménye pl. így nézhet ki:

```
 1 *  *  *  *  *  *  *  *  *  *
 2 *  *  *  *  *  *  *  *  *  *  *
 3 *  *  *  *  *  *  *  *  *  *  *  *  *
 4 *  *  *  *  *  *  *  *
 5 *  *  *  *  *  *  *  *
 6 *  *  *  *  *  *  *
 7 *  *  *  *  *  *  *
 8 *  *  *  *  *  *  *  *  *  *
 9 *  *  *  *  *  *  *  *  *  *  *  *
10 *  *  *  *  *  *  *  *  *  *  *  *  *  *
   1  2  3  4  5  6  7  8  9 10 11 12 13 14
```

8.10. Készítsünk egy szebb hisztogramrajzoló programot. A program generál 100 darab véletlen számot az 1-10 intervallumban, majd ezen véletlen számok előfordulási gyakoriságát megjeleníti egy álló hisztogram formájában. A vízszintes tengelyen az előforduló értékek, míg a függőleges tengelyen ezek gyakorisága szerepel. A gyakoriságot annyi csil-

lag karakter mutatja, ahányszor az adott érték előfordult. A program futási eredménye pl. így nézhet ki:

```
14                                   *
13           *                       *
12           *                   *   *
11       *   *                   *   *
10   *   *   *               *   *   *
 9   *   *   *               *   *   *
 8   *   *   *   *   *       *   *   *
 7   *   *   *   *   *   *   *   *   *   *
 6   *   *   *   *   *   *   *   *   *   *
 5   *   *   *   *   *   *   *   *   *   *
 4   *   *   *   *   *   *   *   *   *   *
 3   *   *   *   *   *   *   *   *   *   *
 2   *   *   *   *   *   *   *   *   *   *
 1   *   *   *   *   *   *   *   *   *   *
     1   2   3   4   5   6   7   8   9  10
```

8.11. Írjuk át leghosszabb kötelet kereső programot úgy, hogy a legrövidebb kötél hosszát is megadja.

8.12. Írjunk egy programot, ami egy bemenetként megadott pozitív egész számról eldönti, hogy prímszám-e vagy nem.
Tipp1: Ellenőrizzük, hogy a szám osztható-e valamelyik nála kisebb, egynél nagyobb számmal. Ha semelyik ilyen számmal nem osztható, akkor biztosan prímszám.
Tipp2: Elegendő csupán a szám gyökénél kisebb számokat ellenőrizni, így sokkal gyorsabb lesz a programunk. Indokoljuk: miért elegendő a szűkített tartományon történő ellenőrzés?

8.13. Az előző prímszám-ellenőrző program felhasználásával írjunk olyan programot, ami egy adott számtartományban (pl. 10000-től 50000-ig) kiírja a prímszámokat.

8.14. Készítsünk programot, ami egy sakktáblát kirajzol az ábra szerinti módon, a sötét mezőket XX, a világosakat két szóköz karakterrel jelölve, a mezők közti határokat jelölve. A program írja fel a tábla szélére a sorok indexeit számokkal, az oszlopokét betűkkel, a sakkban szokásos módon (az A1 mező sötét legyen). A tábla méretét a program olvassa be.

```
kerem a tabla meretet: 6
    A  B  C  D  E  F
  +--+--+--+--+--+--+
 6|  |XX|  |XX|  |XX| 6
  +--+--+--+--+--+--+
 5|XX|  |XX|  |XX|  | 5
  +--+--+--+--+--+--+
 4|  |XX|  |XX|  |XX| 4
  +--+--+--+--+--+--+
 3|XX|  |XX|  |XX|  | 3
  +--+--+--+--+--+--+
 2|  |XX|  |XX|  |XX| 2
  +--+--+--+--+--+--+
 1|XX|  |XX|  |XX|  | 1
  +--+--+--+--+--+--+
    A  B  C  D  E  F
```

9. fejezet

Eljárások, függvények, változók

Ahogy azt az adásvételi szerződések kezelésére írt programunkban láthattuk, a struktúrák használatával az adatszerkezetben természetes módon jelentkező szekvenciális logikai összefüggések a program kódjában is megjelennek. A 6.2. ábrán látható adatszerkezetben pl. a dátum, az eladó, a hajó olyan fogalmak, amelyek több, logikailag összefüggő adatból állnak. Pl. a hajók leírása a nevük, típusuk, hosszuk és árbocszámuk alapján történhet. Ezen logikailag összefüggő adatokat a struktúrák segítségével egyedi adattípussal tudjuk kezelni.

Hasonló logikai összefüggések a program szerkezetében is megfigyelhetők. A 6.3. ábrán pl. a beolvasás és kiírás folyamata is egy-egy logikailag összetartozó egység, de ez – az esetleges megjegyzésektől eltekintve – a program kódjában nem jelentkezik. Azt is megfigyelhetjük, hogy pl. az eladó és vevő beolvasása ugyanolyan típusú tevékenység: egy kalóz adatait kell beolvasni. Ugyanez igaz a különböző dátumok beolvasására is: dátumot olvasunk be a kalózok születési idejének meghatározására, de dátum kell a szerződés keltezéséhez is. Ezen tevékenységek többször is előfordulnak a program során, eddigi kódjainkban azonban nem tudtuk a logikai összetartozást sem megjeleníteni, sem kihasználni. Figyeljük meg, hogy korábbi programunkban pl. a kalóz adatait beolvasó kódrészlet kétszer, a dátum beolvasását végző kódrészlet pedig háromszor jelent meg a kódban.

A logikailag összefüggő szekvenciális programszerkezetek kezelésére az eljárások, illetve függvények használhatók. Ezen túl az eljárások és függvények használatával lehetőségünk lesz a programjainkban használt változók láthatóságának kezelésére, valamint paraméterek átadására is. Ezzel programjaink szerkezete sokkal logikusabb, jobban strukturált lesz.

9.1. Eljárások, függvények

A függvények egymással logikailag összefüggő, szekvenciális tevékenységeket tartalmaznak. A függvények átvehetnek bemenő paramétereket, majd eredményül egy visszatérési értéket adnak vissza. Pl. egy összeadó függvény átveszi az összeadandókat és eredményül visszaadja az összeget. Fontos, hogy a függvény felhasználója (aki pl. az összeadó függvényt programjában meghívja) nem kell, hogy tudjon róla, hogy pontosan *hogyan* oldja meg a függvény a feladatát, csak annyit kell tudnia, hogy *mit* csinál a függvény és mi annak a helyes használati

módja. Nem kell tehát tudni azt, hogy az összeadás a függvényben mi módon lett implementálva, számunkra csak az a fontos, hogy hány darab és milyen típusú számot tud a függvény összeadni és az eredmény milyen típusú lesz. Pl. az összeadó függvény három darab egész számot tud összeadni és az eredmény is egész lesz.

Az eljárások olyan speciális függvények, amelyeknek nincs visszatérési értékük. Pl. egy eljárás lehet egy eredmény formázott kiírása.

Gyakran előfordul az is, hogy egy függvény vagy eljárás nem igényel bemenő paramétert. Pl. ilyen lehet az a függvény, ami visszaadja az aktuális hőmérsékletet. Bemenő paraméter nélküli eljárás lehet például az aktuális időt kiíró eljárás.

A C nyelvben a függvény *deklarációja* azt mondja meg, hogy a függvényt hogyan kell használni: tartalmazza a függvény típusát, nevét, valamit formális paraméterlistáját. Pl. a három egész számot összeadó függvény deklarációja így néz ki:

```
int add(int a, int b, int c);
```

A példában a függvény neve `add`, a függvény visszatérési értékének típusa (röviden: a függvény típusa) `int`, amit a változók típusdefiníciójánál megszokott módon a függvény neve előtt definiálunk. A függvény formális paraméterlistája három bemenő paramétert tartalmaz, az `a`, `b` és `c` paramétereket, esetünkben valamennyi `int` típusú. A deklaráció akkor használatos, ha egy függvényt máshol már definiáltunk, vagy definiálni fogjuk (egyszerű esetekben nem lesz rá szükségünk).

Ahhoz, hogy a függvény valóban végre tudja hajtani feladatát, definiálni kell, hogy milyen műveleteket hajt végre (ez a függvény törzse). Ezt a függvény *definíciójában* adjuk meg. Az összeadó függvényt így definiálhatjuk:

```
int add(int a, int b, int c){
   return a+b+c;
}
```

A függvény *definíciója* a deklarációhoz hasonlóan tartalmazza a függvény fejrészét (típus, név, bemenő paraméterek), valamint a függvény ezt követő törzsében azt is, hogy milyen tevékenységeket hajt végre a függvény. Példánkban a függvény két összeadás műveletet hajt végre, visszatérési értéke a három bemenő paraméter összege lesz. A legtöbb esetben a magunk által készített függvényeket csupán definiáljuk, ilyenkor a deklarációra általában nincs szükség.

A függvényt használatakor nevére hivatkozva kell meghívni, a szükséges bemenő paramétereket zárójelek között kell felsorolni, a visszatérési értéket pedig típusának megfelelően kell felhasználni (pl. értékadáshoz, kiíráshoz). A fenti összeadó függvényt pl. így lehet használni:

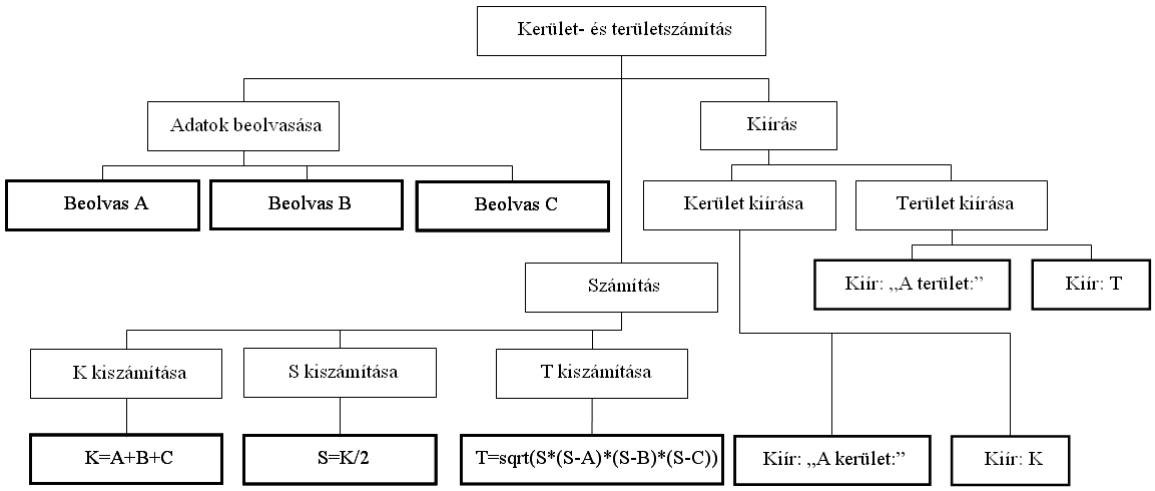
```
int s1, s2, s3, sum;
s1=1; s2=2; s3=3;
sum=add(s1,s2,s3);
printf("4+8+7=%d\n", add(4,8,7));
```

Az átadott aktuális paraméterlistán szereplő paraméterek (`s1`, `s2`, `s3`) típusa példánkban megegyezik a függvény paraméterlistáján látható típusokkal. A visszaadott egész típusú értéket a példában egy egész típusú változóba töltöttük, illetve egész számként írtuk ki. Amennyiben a paraméterek átadásakor az aktuális paraméter típusa különbözik a formális paraméter típusától, akkor a fordító megpróbál egy „ésszerű” konverziót végrehajtani. Pl. ha egy lebe-

gőpontos számot adunk át egész helyett, akkor csonkolja azt (pl. 3.1415-ből 3 lesz). Ez a mechanizmus az értékadás műveletekre is hasonlóan működik.
Más programozási nyelvekben azon függvényeket, amelyek nem adnak vissza visszatérési értéket, eljárásnak nevezik. A C nyelvben is vannak ilyen tulajdonságú függvények, ezeknek típusát a `void` kulcsszó jelöli: a void függvények ekvivalensek az eljárásokkal.

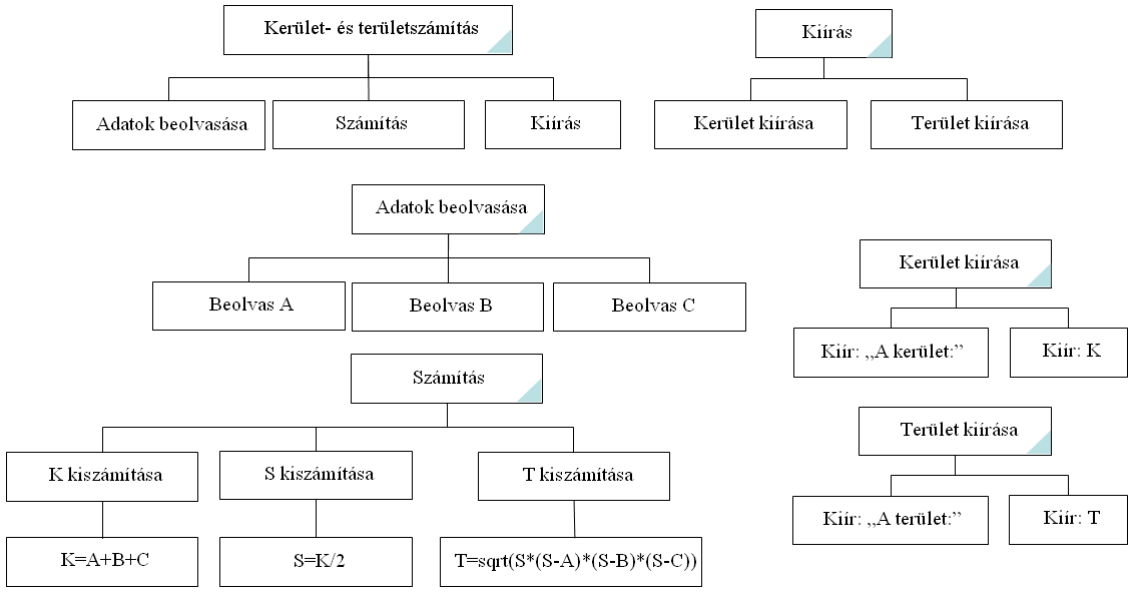
9.1. példa: Írjuk át a háromszög területét és kerületét számító programot függvények használatával [*9.haromszog_fv.c*].

A 6.1. ábrán látható terület és kerületszámító programot egyszerű szekvenciális tevékenységszerkezettel már megoldottuk. Az a program azonban nem tükrözte az ábrán jól látható logikai összefüggéseket az egyes tevékenységek között. Nem különültek el a beolvasás, a számítás és a kiírás műveletek egymástól. A programkódban – hacsak megjegyzéseket nem helyezünk el benne – nem jelennek ezek a fogalmak, ott csak az elemi műveleteket találjuk meg egymás után. Valójában az egyszerű szekvenciális programunk a 6.1. ábrán látható fastruktúra végpontjait, leveleit tartalmazza, a közbülső pontokon megjelenő fogalmakat nem (nincs a programban pl. Adatok beolvasása, csak az egyes élek hosszainak beolvasásának szekvenciáját találjuk meg a kódban egyéb utasítások között, hasonlóan nincs Számítás, csak a beolvasó utasításokat követi néhány elemi aritmetikai művelet, stb.). A 9.1. ábra mutatja a 6.1. ábra kissé átrajzolt vezérlési szerkezeteit, külön jelölve a faszerkezet leveleit, amelyek az egyszerű programban elemi utasításként jelentek meg. A program utasításainak sorrendje a Jackson-ábra leveleinek sorrendjével egyezik meg, balról jobbra olvasva.



9.1. ábra. A terület- és kerületszámoló program szerkezete, ahol az elemi műveleteket (a fa leveleit) vastag keretek jelölik. Az egyszerű szekvenciális program ezen utasításokat tartalmazza (balról jobbra olvasva).

Alakítsuk át ezt a programot most úgy, hogy abban megjelenjenek a magasabb szintű fogalmak, melyek logikailag összefüggő tevékenységeket fognak össze. A programot leíró Jackson-ábrát kissé átrajzolt változatát mutatja a 9.2. ábra, ahol most minden fontos összetett tevékenység kifejtését külön felrajzoltuk. Az egyes összetett tevékenységek lesznek majd az eljárások vagy függvények, ezek kifejtése pedig az eljárás/függvény definíciója. Pl. a Kiírás eljárás definíciója tartalmazza, hogy ez az eljárás két további (összetett) művelet szekvenciáját tartalmazza: a Kerület kiírása és a Terület kiírása műveleteket, amelyek maguk is eljárások lesznek.



9.2. ábra. A terület- és kerületszámoló program szerkezete, ahol az eljárások definíciója most elkülönülten látható. Az eljárások definíciói a színes sarkokkal jelöl téglalapoknál kezdődnek.

A Jackson ábrán a program definíciója a Kerület- és területszámítás tevékenységnél kezdődik. Ennek szerepét a C programban a main függvény veszi át: minden C program végrehajtása itt kezdődik. A többi jelölt összetett tevékenységnek egy-egy függvényt fogunk megfeleltetni. A függvények nevei a C szintaxisnak megfelelően nem tartalmazhatnak ékezetes karaktert és szóközt sem, de a programban a 9.2. ábra neveihez hasonló függvényneveket fogunk használni.

Mielőtt egy C programban egy függvényt használni lehetne egy másik függvény definíciójában, a felhasznált függvényt definiálni vagy legalább deklarálni kell. Ezért az egyszerűbb C programokban hátul áll a main függvény, és a függvények definícióját olyan sorrendben készítjük el, hogy a hivatkozott függvények előtte már definiálva legyenek. Tehát ha A függvényben használjuk B függvényt, akkor B függvényt előbb definiáljuk, mint A-t. Ha a függőségek nem tartalmaznak kört (pl. A használja B-t és B használja A-t), akkor ez az út mindig járható. Abban az esetben, ha nem tudunk megfelelő sorrendet találni, akkor a C fordító azt is megengedi, hogy a függvényt először deklaráljuk, ezzel hivatkozhatóvá válik, majd később definiáljuk. Pl. deklaráljuk A-t, definiáljuk B-t (itt már hivatkozhatunk A-ra), majd definiáljuk A-t (itt pedig már hivatkozhatunk B-re). Egyszerű példáinkban erre nem lesz szükség.

A C nyelv standard függvényeinek deklarációi az include sorokban megadott állományokban vannak (pl. a printf függvény deklarációját az stdio.h fájl tartalmazza). Ezért az #include direktívákat a program elejére írva biztosítjuk, hogy az összes standard függvény a programunkban hivatkozható lesz.

Megjegyzés: A fordító (compiler) számára csak a felhasznált függvények deklarációja fontos, hiszen ennek segítségével ellenőrizni tudja, hogy helyesen használjuk-e a függvényt, illetve a hívási és visszatérési paraméterek átadását (esetleges típuskonverziókkal együtt) meg tudja valósítani. Természetesen a futtatható kód elkészítéséhez szükség lesz valamennyi függvény definíciójára is: ezt az információt a szerkesztő (linker) használja. Ezért kaphatunk mind a fordítótól, mind a szerkesztőtől hibaüzeneteket a program fordítási folyamata közben.

Korábbi programjainkban egyetlen függvényt (a main-t) használtunk csak. Ezekben a programokban a felhasznált változókat néha a függvény előtt definiáltuk, máskor a függvényen belül, de ennek különösebb jelentőséget eddig nem tulajdonítottunk. Amennyiben több függvényt használunk, ennek nagyon nagy jelentősége lesz.

Amennyiben programunkban a változó definícióját a függvények definíciói előtt (a függvényeken kívül) helyeztünk el, akkor globális változóról beszélünk: ezek minden függvény belsejéből „láthatók”, függvényeinkből ezeket szabadon írhatjuk és olvashatjuk. A globális változók nagyon egyszerű lehetőséget adnak tehát függvényeink között az adatok átadására: az egyik függvény írja a változót, a másik pedig később olvassa azt. Ez a megoldás azonban nagyobb programok esetén a globális változók elszaporodásához vezet, ami egy idő után áttekinthetetlen programkódot eredményez. További hátránya a globális változóknak az, hogy ezeket olyan függvények is láthatják, amelyeknek nincs hozzájuk köze. A globális változók használatát lehetőségek szerint kerülni kell.

A legtöbb programozási nyelvben lehet lokális változókat is használni. Ezen változók csak bizonyos eljárásokból, függvényekből láthatók. A C nyelvben igen egyszerű a lokális változók kezelése: a lokális változó csak egyetlen függvényben látható, mégpedig abban, amelyikben deklaráltuk. A lokális változókat általában a függvény törzsének elején szokás deklarálni, pl. így:

```
void Boo(){
     int A;  /* lokális változó, csak a Boo függvényben látszik */
     ... /* utasítások*/
}
```

Megjegyzés: változót lehet később is deklarálni, erre az a szabály vonatkozik, hogy csak olyan változóra lehet a kódban hivatkozni, amit korábban már deklaráltunk. Sőt a C nyelv megengedi a blokkon belüli lokális változók használatát is, amelyek csak az adott blokkban láthatóak.

A változók láthatósági kérdéseivel részletesebben a 9.2. fejezetben foglalkozunk majd.

Lássuk tehát programunkat most függvények használatával megvalósítva úgy, hogy nem használunk globális változókat [*9.haromszog_fv.c*]:

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

/* a háromszög adatainak ábrázolásához használt struktúra */
struct haromszog{
     double A; /* A oldal */
     double B; /* B oldal */
     double C; /* C oldal */
};
```

```
/* az eredmények tárolására használt struktúra */
struct eredmeny{
      double K; /* kerület */
      double T; /* terület */
};

/*
  Adatok beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – háromszög adatai
 */
struct haromszog AdatokBeolvasasa(){
      struct haromszog H;
      printf("Az A oldal hossza:");
      scanf("%lf", &H.A);
      printf("A B oldal hossza:");
      scanf("%lf", &H.B);
      printf("A C oldal hossza:");
      scanf("%lf", &H.C);
      return(H);
}

/*
  Kerület és terület számítása
  bemenet: paraméter - háromszög adatai
  kimenet: visszatérési érték – eredmény (kerület és terület)
 */
struct eredmeny Szamitas(struct haromszog H){
      double S;
      struct eredmeny E;
      E.K=(H.A+H.B+H.C);                    /* kerület */
      S=E.K/2;                              /* fél kerület */
      E.T=sqrt(S*(S-H.A)*(S-H.B)*(S-H.C)); /* terület */
      return E;
}
```

```
/*
  Kerület kiírása
  bemenet: paraméter - háromszög kerülete
  kimenet: standard kimenet
 */
void KeruletKiirasa(double K){
     printf("\nA kerulet: %lf\n", K);
}

/*
  Terület kiírása
  bemenet: paraméter - háromszög területe
  kimenet: standard kimenet
 */
void TeruletKiirasa(double T){
     printf("\nA terulet: %lf\n", T);
}

/*
  Háromszög kerületének és területének kiírása
  bemenet: paraméter - eredmény adatok
  kimenet: standard kimenet
 */
void Kiiras(struct eredmeny E){
     KeruletKiirasa(E.K);
     TeruletKiirasa(E.T);
}

/*
  Háromszög adatainak bekérése, kerület és terület számítása és
  kiírása
  bemenet: standard bemenet
  kimenet: standard kimenet
 */
int main(){
     struct haromszog Haromszog;
     struct eredmeny Eredmeny;
     Haromszog=AdatokBeolvasasa();
     Eredmeny=Szamitas(Haromszog);
     Kiiras(Eredmeny);
     return 0;
}
```

Példánkban nincsenek globális változók, minden változó csak ott létezik és látható, ahol arra szükség van. A `main` függvényben deklaráljuk a `Haromszog` és az `Eredmeny` változókat. Az `AdatokBeolvasasa` függvény belsejében létrehozunk egy `H` nevű haromszog típusú struktúrát, amelybe a beolvasó függvény meghívása után a korábbi módon betölti a háromszög adatait. (Ez a `H` nevű változó csak ebben a függvényben létezik és csak itt látható: amikor a függvényt meghívjuk, létrejön a változó, amikor a függvény véget ér, a változó megszűnik.) A függvény végén a `H` értékét visszaadjuk, így az a main függvény `Haromszog` nevű változójának adódik értékül. Ezután a main függvény a `Szamitas` függvényt hívja meg a `Haromszog` paraméterrel, majd a számítás eredménye az `Eredmeny` változóba töltődik be. A Szamitas függvény indításakor a Haromszog változó értéke (bemenő paraméter) a függvény belsejében a H nevű paraméternek adódik értékül. (Ez a H paraméter természetesen nem azonos az `AdatokBeolvasasa` függvény hasonló nevű paraméterével: mivel mindkettő lokális paraméter, ezek egymásról mit sem tudnak, csupán a nevük azonos. Hogy éppen melyik változóról van szó, azt az dönti el, hogy melyik függvényben hivatkozunk erre a névre). A `Szamitas` függvény a számítást a `H` adatainak segítségével végzi el, majd az eredményt egy `eredmeny` típusú, `E` nevű változóba tölti. A függvény ennek a változónak az értékét adja vissza a `return` utasítással. A `main` függvényben ekkor az `Eredmeny` nevű változónak értékül adódik az `E` értéke (majd a `Szamitas` függvény befejezése után `E` megszűnik). Hasonlóan a `Kiiras` függvény átveszi az `Eredmeny` értékét, amire a függvényen belül `E` névvel hivatkozik (ami természetesen más `E` nevű változó, mint a `Szamitas` függvénybeli). A kiírás meghívja a területet és kerületet kiíró függvényeket, mindegyiket a megfelelő lebegőpontos bemenő adattal. Figyeljük meg, hogy minden függvényt a neki releváns bemenő adattal hívunk meg: például a `KeruletKiirasa` függvényt a `Kiiras` függvény meghívhatta volna a teljes `E` eredmény struktúrával is, aminek `K` mezőjét kiírhatta volna a `KeruletKiirasa`, de ez helytelen megoldás lett volna: a `KeruletKiirasa` függvény számára a kerület (egy szám) a releváns adat és nem a területet is tartalmazó teljes eredmény (ami egy struktúra).

Vizsgáljuk meg programunkban a függvények definíciójának sorrendjét. A Kiiras függvény tartalmazza a kerület és a terület kiírását végző függvény hívását, míg a többi függvény csak a standard függvénykönyvtárakat alkalmazza. Figyeljük meg, hogy a példában a main függvény definíciója áll a kód végén (hiszen az használja az AdatokBeolvasasa, Szamitas, és Kiiras függvényeket), míg a Kiiras függvény definícióját meg kell előzze a KeruletKiirasa és a TeruletKiirasa függvények definíciója.

9.2. példa: Próbáljuk meg a kódban felcserélni pl. a Kiiras és TeruletKiirasa függvények definíciójának sorrendjét. A fordító hibaüzenetet ad. Most helyezzük el a Kiiras definíciója elé a TeruletKiirasa függvény deklarációját. Kódunk ezen részlete tehát így fog kinézni:

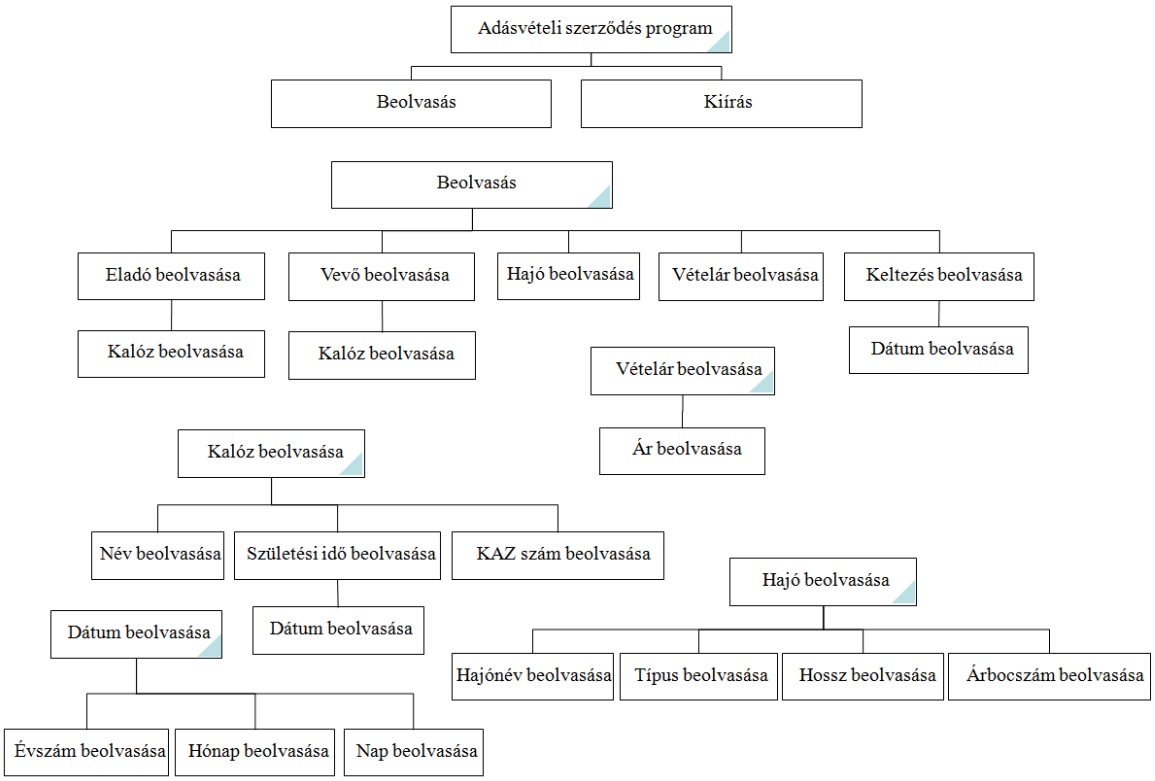
```
/* itt deklaráljuk a TeruletKiirasa függvényt: */
void TeruletKiirasa(double T);

void Kiiras(struct eredmeny E){
   KeruletKiirasa(E.K);
   TeruletKiirasa(E.T); /*itt már hivatkozunk a deklarált függvényre*/
}
```

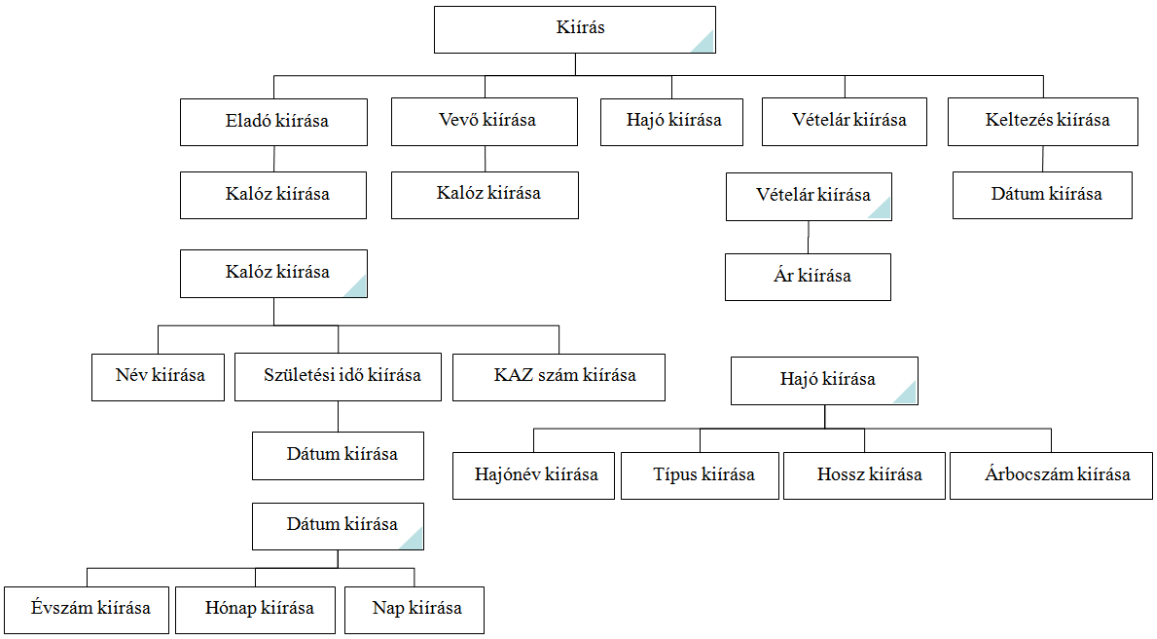
```
/* itt definiáljuk a TeruletKiirasa függvényt: */
void TeruletKiirasa(double T){
   printf("\nA terulet: %lf\n", T);
}
```

Ez a kód már hibátlanul lefordul és helyesen fut.

9.3. példa: Alakítsuk át a hajós adásvétel programot úgy, hogy függvényeket használjon. Ne használjunk gloális változókat, ügyeljünk arra, hogy minden függvény csak a számára fontos információkhoz férjen hozzá.

A 6.3. ábrát kicsit átrajzolva láthatjuk a 9.3. ábrán, ahol a függvények definícióinak kezdetét színes sarkokkal jelöltük. Figyeljük meg, hogy egyes függvényeket több helyen is használjuk: pl. a Kalóz beolvasása függvényt a Beolvasás függvény kétszer is meghívja, a Dátum beolvasása pedig a program során háromszor is megtörténik (az eladó, a vevő és a keltezés beolvasásánál). Természetesen ezeket a függvényeket is csak egyszer kell definiálni (ahogy az ábrán is látható).





9.3. ábra. A hajók adásvételi szerződését kezelő program 6.3. ábrán látható leírásának kissé módosított változata. A színes sarkú téglalapok az eljárások definíciójának kezdetét mutatják.

Az ábrán összesen kilenc eljárást látunk, ezek az Adásvételi szerződés program, Beolvasás, Kiírás, Kalóz beolvasása, Hajó beolvasása, Dátum beolvasása, Kalóz kiírása, Hajó kiírása és Dátum kiírása. A C nyelvű programban is ezek a függvények jelennek meg, hasonló neveken. Az Adásvételi szerződés program szerepét a main függvény veszi át. A C nyelvű program függvényekkel megvalósított változata tehát így nézhet ki [*9.adasvetel_fv.c*]:

```
/*
* Hajó adásvételi szerződés adatainak bekérése és kiírása függvé-
nyekkel.
 */

#include <stdio.h>
#define _MAX_HOSSZ 20

struct datum{
     unsigned int Evszam;  /* a dátum évszám mezője */
     unsigned int Honap;   /* a dátum hónap mezője */
     unsigned int Nap;     /* a dátum nap mezője */
};

struct kaloz{
     char Nev[_MAX_HOSSZ+1];          /* a kalóz neve */
     struct datum Szuletesi_ido;      /* a kalóz születési ideje */
     unsigned int KAZ_szam;           /* a kalóz KAZ-száma */
};
```

```
struct hajo{
     char Nev[_MAX_HOSSZ+1];          /* a hajó neve */
     char Tipus[_MAX_HOSSZ+1];        /* a hajó típusa */
     unsigned int Hossz;              /* a hajó hossza */
     unsigned int Arbocszam;          /* a hajó árbocainak száma */
};

struct szerzodes{
     struct kaloz Elado;         /* az eladó adatai */
     struct kaloz Vevo;          /* a vevő adatai */
     struct hajo Hajo;           /* a hajó adatai */
     unsigned int Vetelar;       /* a hajó vételára */
     struct datum Keltezes;      /* a szerződés kelte */
};


/*
  Dátum beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – dátum
 */
struct datum DatumBeolvas(char szoveg[]){
     struct datum Datum;
     printf("%s", szoveg);
     scanf("%d.%d.%d.", &Datum.Evszam, &Datum.Honap, &Datum.Nap);
     return Datum;
}


/*
  Dátum kiírása
  bemenet: paraméter - dátum
  kimenet: standard kimenet
 */
void DatumKiir(char szoveg[], struct datum Datum){
     printf("%s%d.%d.%d.\n", szoveg, Datum.Evszam, Datum.Honap,
Datum.Nap);
}
```

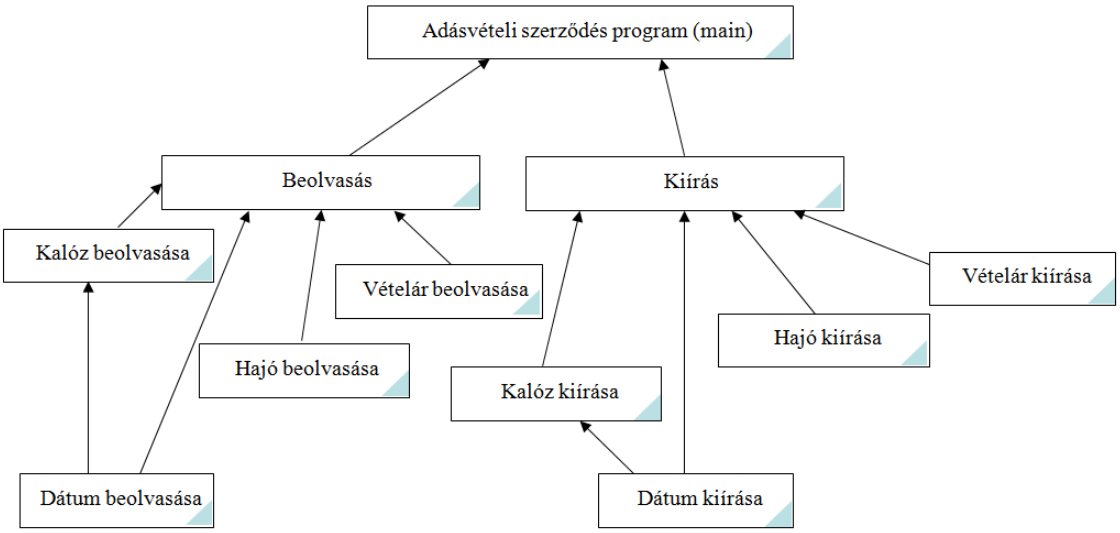
```
/*
  Kalóz adatainak beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – kalóz adatai
 */
struct kaloz KalozBeolvas(char szoveg[]){
     struct kaloz Kaloz;
     printf("%s\t\tNev: ", szoveg);
     scanf("%s", Kaloz.Nev);
     Kaloz.Szuletesi_ido=DatumBeolvas("\t\tSzuletesi ido [e.h.n.]: ");
     printf("\t\tTitkos kaloz azonosito: ");
     scanf("%d", &Kaloz.KAZ_szam);
     return Kaloz;
}
/*
  Kalóz adatainak kiírása
  bemenet: paraméter - kalóz adatai
  kimenet: standard kimenet
 */
void KalozKiir(char szoveg[], struct kaloz Kaloz){
     printf("%s\t\tNev: %s\n", szoveg, Kaloz.Nev);
     DatumKiir("\t\tSzuletesi ido: ", Kaloz.Szuletesi_ido);
     printf("\t\tTitkos kaloz azonosito: %d\n", Kaloz.KAZ_szam);
}
/*
  Hajó adatainak beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – hajó adatai
 */
struct hajo HajoBeolvas(){
     struct hajo Hajo;
     printf("\n\tHajo adatai:\n");
     printf("\t\tNev: ");
     scanf("%s", Hajo.Nev);
     printf("\t\tTipus: ");
     scanf("%s", Hajo.Tipus);
     printf("\t\tHajo hossza: ");
     scanf("%d", &Hajo.Hossz);
     printf("\t\tHajo arbocszama: ");
     scanf("%d", &Hajo.Arbocszam);
     return Hajo;
}
```

```
/*
  Hajó adatainak kiírása
  bemenet: paraméter - hajó adatai
  kimenet: standard kimenet
 */
void HajoKiir(struct hajo Hajo){
     printf("\n\tHajo adatai:\n");
     printf("\t\tNev: %s\n", Hajo.Nev);
     printf("\t\tTipus: %s\n", Hajo.Tipus);
     printf("\t\tHajo hossza: %d\n", Hajo.Hossz);
     printf("\t\tHajo arbocszama: %d\n", Hajo.Arbocszam);
}

/*
  Vételár beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – vételár
 */
unsigned int ArBeolvas(){
     unsigned int Ar;
     printf("\n\tHajo vetelara: ");
     scanf("%u", &Ar);
     return Ar;
}

/*
  Vételár kiírása
  bemenet: paraméter - vételár
  kimenet: standard kimenet
 */
void ArKiir(unsigned int Ar){
     printf("\n\tHajo vetelara: %d\n", Ar);
}
```

```
/*
  Szerződés összes adatának beolvasása
  bemenet: standard input
  kimenet: visszatérési érték - szerződés adatai
 */
struct szerzodes Beolvas(){
   struct szerzodes Szerzodes;
   printf("Kaloz adasveteli szerzodes bevitele\n");
   Szerzodes.Elado=KalozBeolvas("\n\tElado adatai:\n");
   Szerzodes.Vevo=KalozBeolvas("\n\tVevo adatai:\n");
   Szerzodes.Hajo=HajoBeolvas();
   Szerzodes.Vetelar=ArBeolvas();
   Szerzodes.Keltezes=
               DatumBeolvas("\n\tSzerzodeskotes idopontja [e.h.n.]: ");
   return Szerzodes;
}

/*
  Szerződés összes adatának kiírása
  bemenet: paraméter - szerződés adatai
  kimenet: standard kimenet
 */
void Kiir(struct szerzodes Szerzodes){
     printf("\n\nKaloz adasveteli szerzodes adatainak listazasa\n");
     KalozKiir("\n\tElado adatai:\n", Szerzodes.Elado);
     KalozKiir("\n\tVevo adatai:\n", Szerzodes.Vevo);
     HajoKiir(Szerzodes.Hajo);
     ArKiir(Szerzodes.Vetelar);
     DatumKiir("\n\tSzerzodeskotes idopontja: ",
Szerzodes.Keltezes);
}

/*
  Szerződés adatainak beolvasása és kiírása
  bemenet: paraméter - szerződés adatai
  kimenet: standard kimenet
 */
int main(){
     struct szerzodes Szerzodes;
     Szerzodes=Beolvas();
     Kiir(Szerzodes);
     return 0;
}
```

A main függvény mos csak egy lokális változót, a `Szerzodes` nevű struktúrát tartalmazza, valamint a `Beolvas` és a `Kiir` függvények hívását. A beolvasott szerződés adatait a `Beolvas` függvény a `Szerzodes` valtozóba tölti, majd ezen változóval, mint bemenő paraméterrel hívjuk meg a `Kiir` függvényt.

A függvények egymás közötti függőségeit a 9.4. ábrán látható függőségi gráfon ábrázolhatjuk: a felhasznált függvényekből nyíl mutat a használó függvény felé. Pl. a Dátum beolvasása függvényt használja mind a Kalóz Beolvasása, mind a Beolvasás függvény. Ebből a függvények definíciójának sorrendje meghatározható: olyan sorrendet kell használni, amelyben a felhasznált függvényt mindig a használat előtt először kell definiálni. Például a C forráskódban a `Beolvas` és `Kiir` függvényeknek a main függvény előtt kell állnia, a `Beolvas` függvényt meg kell előzzék a KalozBeolvas, HajoBeolvas, ArBeolvas és DatumBeolvas függvények, stb. A fenti program szerkezete figyelembe veszi ezen ezen függőségeket. Természetesen ezzel a problémával nem kell foglalkoznunk, ha a függvényeket használatuk előtt (tipikusan a kód elején) deklaráljuk: ekkor a függvények definíciói tetszőleges sorrendben állhatnak.



9.4. ábra. A hajók adásvételi szerződését kezelő program függvényeinek függőségi gráfja. A nyilak a használt függvényből mutatnak a használó függvény felé

Figyeljük meg a lokális változók és paraméterek használatát. Minden függvény csak a számára fontos adatokat látja: a `main` függvény látja a teljes szerződést, a `Beolvas` szintén a teljes szerződést látja (és azt adja vissza visszatérési értékül), míg pl. a `KalózBeolvas` csak az aktuális kalóz (eladó vagy vevő) adatait látja és adja vissza. Hasonlóan pl. a `DatumBeolvas` csak a beolvasandó dátumot kezeli. A `DatumBeolvas` függvényben megfigyelhetjük, hogy egy karaktertömb bemenő paramétert vesz át, ebben adja át a hívó, hogy mit írjon ki a beolvasás előtt (születési idő vagy a szerződéskötés időpontja).

A C nyelvben a tömböket átadhatjuk a fenti programban látott módon, pl.:

```
struct kaloz KalozBeolvas(char szoveg[])
```

Itt egy `szoveg` nevű, karakterekből álló tömböt adunk át bemenő paraméterül. Azt, hogy tömbről, és nem egyetlen karakterről van szó, a változó neve mögötti szögletes zárójelpár jelzi. Figyelem: itt nem kell megadni a tömb méretét, csupán azt a tényt jelezzük a fordítónak, hogy itt egy tömbről van szó (erről részletesebben a 10. fejezetben lesz még szó).

A C nyelvben a változókat érték szerint adjuk át. A `Kiir` függvényben pl. így hívjuk meg az `ArKiir` függvényt:

```
ArKiir(Szerzodes.Vetelar);
```

Az `Arkiir` függvény deklarációja pedig így néz ki:

```
void ArKiir(unsigned int Ar)
```

Az `ArKiir` függvény meghívásakor a `Szerzodes.Vetelar` (egész típusú) változó értéke átmásolódik az `ArKiir` függvény bemenő paraméterébe, aminek neve `Ar`. Az `ArKiir` függvényen belül nem látszik a `Szerzodes.Vetelar` változó, csak annak `Ar` nevű másolata. (Ha pl. az `ArKiir` függvényben átírnánk az `Ar` értékét, annak természetesen semmi hatása nem lenne a hívó függvényben található `Szerzodes.Vetelar` változó értékére.) Ezt a módszert hívjuk érték szerinti paraméterátadásnak.

Ez alól a mechanizmus alól egy kivétel van a C nyelvben: a tömböket nem érték szerint adjuk át, hanem csupán a tömbök *címei* kerülnek (érték szerinti) átadásra. Ezt a megoldást más nyelvekben cím szerinti átadásnak nevezik. Ebben az esetben nem másolódik át a változó tartalma a meghívott függvény lokális paraméterébe, hanem csupán a változó címét adjuk át. Ezzel – különösen a nagy méretű paraméterek átadása során – időt és memóriát lehet spórolni. Fontos jellemzője a cím szerinti átadásnak, hogy minden változtatás, amit a hívott függvényben a cím szerint átadott változón végrehajtunk, természetesen a hívó függvényben is megjelenik – hiszen ugyanarról a változóról van szó, nem jött létre másolat. Tehát a

```
Szerzodes.Vevo=KalozBeolvas("\n\tVevo adatai:\n");
```

függvényhívás során a karakterlánc a hívó – Beolvas nevű – függvényben tárolódik és csak ezen karakterlánc címe adódik át, nem pedig a teljes karakterlánc.

A fentebb leírtak nem csak a paraméterek átadására, hanem általában az értékadásra is jellemzők. Az `a = b` utasítás általában azt jelenti, hogy az `a` változóba átmásolódik a `b` értéke. Ez a tömbök kivételével valóban új kópia létrejöttét jelenti, a tömbök esetén azonban maga a tömb nem másolódik át, csupán annak címe.

A tömbök eme „furcsa” viselkedésre már láttunk példát a `scanf` függvény használatakor is: ha pl. egy egész típusú változót olvasunk be, akkor a változó neve elé kell a & címoperátor, míg a tömbök neve elé nem kell (ill. nem szabad) kitenni a címoperátort. A C nyelvben tehát a tömb neve nem a tömb értékét jelenti, hanem a tömb kezdőcímét, ellentétben az egyszerű változókkal és a struktúrákkal, ahol a változó neve a változó értékét jelöli. A különleges bánásmódot a nyelv tervezői azért alkalmazták, hogy – a potenciális nagy méretű – tömbök átadása, értékadása is gyors művelet legyen. Ez a legtöbb esetben valóban jó megoldás, ha pedig tömböket mégis másolni szeretnénk, akkor erre külön függvényeket kell alkalmazni.

Mivel tömböt nem lehet érték szerint átadni, így a függvények sem tudnak tömböt visszaadni. Ennek megoldása a C nyelvben kétféleképpen történhet:

1. A visszaadni kívánt tömböt egyszerűen egy struktúrába ágyazzuk.
2. A visszaadni kívánt tömböt a függvény hívása előtt, a hívó függvényben létrehozzuk, ezt a tömböt bemenő paraméterként átadjuk (amely átadás a tömb címét jelenti), majd a függvény belsejében kitöltjük a tömböt. Ekkor a függvénynek nem kell „valódi” visszatérési értéket adnia, hiszen az egyik bemenő paraméteren megjelenő „mellékhatás” a visszatérési érték.

9.4. példa: Írjunk programot, amelyben egy függvény átvesz egy karaktertömböt, majd minden elemét megnöveli eggyel. A visszaadott tömb legyen struktúrába ágyazva [*9.tombfv1.c*].

Az `f1` függvénynek bemenő paraméterül megadjuk az `X` tömböt, valamint a tömbben található karakterek `N` számát. A függvény egy olyan `tombstruct` típusú struktúrával tér vissza, amelyben egy 10 hosszúságú karakter tömb van:

```
#include <stdio.h>

struct tombstruct{
   char tomb[10];   /*egyetlen tömböt tartalmazó struktúra*/
};

/*
  Tömbön operáló példafüggvény (tömb minden elemét megnöveli)
  bemeneti paraméterek:
      X: bemenő string címe
      N: a string hossza
  kimenet: visszatérési érték - stringet tartalmazó struktúra
 */
struct tombstruct f1(char X[], int N){
   int i;
   struct tombstruct Y;  /* ebben tároljuk a számított értéket*/
   for (i=0; i<N;i++){
      Y.tomb[i]=X[i]+1;  /* tömbön végzett művelet */
   }
   Y.tomb[N-1]=0;        /* stringet lezáró 0 hozzáadása*/
   return Y;             /* visszatérési érték */
}

int main(){
   char T[10]="ABCDEFGHI";   /* ez lesz a bemenő paraméter*/
   struct tombstruct A;      /* ebben tároljuk a kimenetet*/

   printf("elotte: %s\n", T);       /* bemenet kiírása */
   A=f1(T,10);                      /* függvény hívása */
   printf("utana:  %s\n", A.tomb);  /* kimenet kiírása */
   return 0;
}
```

9.5. példa: Írjuk át a fenti programot úgy, hogy a visszaadott tömb címe egy bemenő paraméter legyen [*9.tombfv2.c*].

Az `f2` void függvénynek bemenő paraméterül megadjuk az `X` tömböt (illetve annak címét), amely a bemenő karaktersorozatot tartalmazza, az `Y` tömböt, amelyben az eredményt adjuk vissza, valamint a tömbben található karakterek `N` számát. Figyeljük meg, hogy a hívó

`main` függvényben létrehozzuk mind a bemenő `T`, mind a kimenő `W` tömböt. Ezeket (illetve, mint tudjuk: címüket) adjuk át az `f2` függvénynek. Az eredmény a `W` tömbben képződik:

```
#include <stdio.h>

/*
  Tömbön operáló példafüggvény (tömb minden elemét megnöveli)
  bemeneti paraméterek:
      X: bemenő string címe
      X: eredmény string címe
      N: a string hossza
  kimenet: -
 */
void f2(char X[], char Y[], int N){
   int i;
   for (i=0; i<N;i++){
      Y[i]=X[i]+1;
   }
   Y[N-1]=0; /*stringet lezáró 0*/
}

int main(){
   char T[10]="ABCDEFGHI";  /* bemenő karaktersorozat */
   char W[10];              /* ez lesz a kimenet*/

   printf("elotte: %s\n", T);   /* bemenet kiírása */
   f2(T, W, 10);                /* függvény hívása */
   printf("utana:  %s\n", W);   /* kimenet kiírása */
   return 0;
}
```

Mindkét program a következő futási eredményt adja:

```
elotte: ABCDEFGHI
utana:  BCDEFGHIJ
```

9.2. Változók láthatósága és élettartama

Programjainkban a változók láthatósági tartománnyal rendelkeznek: minden változó a saját láthatósági tartományán belül látható, elérhető, használható. A globális változók láthatósági tartománya kiterjed valamennyi függvényre, eljárásra, tehát a globális változót a program valamennyi utasításából elérhetjük; más változók ellenben csak szűkebb körben láthatók. Egyes

nyelvek megengedik, hogy függvények/eljárások egyes csoportjaiban legyen látható egy változó, a C nyelvben azonban ennél egyszerűbb modellt alkalmazunk: egy változó vagy globális (az adott program minden függvényében látható), vagy egyetlen függvényben látható csupán (lokális az adott függvényben, mégpedig abban a függvényben, ahol deklaráltuk). A lokális változók eltakarhatnak globális változókat, ha azokat ugyanolyan néven újradeklaráljuk. A függvények bemenő paraméterei is lokális változóként foghatók fel. A C nyelv ezen kívül lehetővé teszi, hogy a függvényeken belüli blokkokban is definiálhassunk – csak az adott blokkban látható – változókat.

9.6. példa: Az alábbi programban az i, j és a változóneveket használjuk mind globális, mind lokális változóként, sőt paraméterátadáskor is. Vizsgáljuk meg a program működését [*9.valtozok1.c*].

```
1.   #include <stdio.h>
2.   int i=1, j=5;
3.
4.   int f1(int j){
5.      printf("f1: i=%d, j=%d\n", i, j);     /* itt i=1 és j= 3*/
6.      j=7;
7.      printf("f2: i=%d, j=%d\n", i, j);     /* itt i=1 és j= 7*/
8.      return(j);
9.   }
10.
11.  int main(){
12.     int i=2, j=3;
13.     printf("m1: i=%d, j=%d\n", i, j);     /* itt i=2 és j=3*/
14.     { /* a blokk kezdete */
15.        int i=4;
16.        printf("m2: i=%d, j=%d\n", i, j);  /* itt i=4 és j=3*/
17.        i=f1(j);
18.        printf("m3: i=%d, j=%d\n", i, j);  /* itt i=7 és j=3*/
19.     } /* a blokk vége */
20.     printf("m4: i=%d, j=%d\n", i, j);     /* itt i=2 és j=3*/
21.     return 0;
22.  }
```

A program kimenete a következő:

```
m1: i=2, j=3
m2: i=4, j=3
f1: i=1, j=3
f2: i=1, j=7
m3: i=7, j=3
m4: i=2, j=3
```

A 2. sorban két globális változót, az i és j nevű egészeket definiáljuk. Itt kezdeti értéket is adunk nekik, az i értéke 1, a j értéke 5 lesz. Ezen változók a program (pontosabban a forrásfájl) minden függvényében látszanak. A main függvényben (12. sor) is létrehozunk két változót, amelyeknek neve szintén i és j. Ezen változók lokálisak, csak a main függvényben láthatók. Mivel a lokális változók nevei megegyeznek a globális változók neveivel, így ezek *eltakarják* a globális változókat: a main függvényben az i és j nevű globális változók már nem látszanak, ezek helyett az azonos nevű lokális változókat látjuk, amelyeknek kezdeti értékei 2 és 3. A 13. sor kiíró utasítása a lokális változók értékét fogja kiírni.

A main függvény 14. és 19. sora között egy blokkot definiálunk egy kapcsos zárójelpár között. Ilyen blokkokkal már találkoztunk az elágazások és ciklusok használata közben, de blokkot bárhol definiálhatunk. Itt a blokk kezdetén létrehoztunk egy i nevű változót (15. sor). Ez a változó csak a blokkon belül lesz látható, és mivel azonos nevű a korábban definiált függvényben lokális i változóval, így eltakarja azt (természetesen a globális i változót is). A blokkban tehát a 15. sorban definiált, 4 kezdeti értékű i változó látható. Természetesen látható itt is a függvényben lokális j nevű változó, amint a 16. sor kiíró utasításában láthatjuk.

Az f1 nevű függvény bemenő paraméterének neve j, tehát az j nevű változó az f1 függvényben a bemenő paraméter értékét tartalmazza és eltakarja a globális j nevű változót. A globális i változó természetesen látszik az f1 függvényben is. A függvényt a main 17. sorában hívjuk meg a j (main-ben lokális változó) értékével, ami a híváskor 3. Ezen érték átmásolódik az f1-ben lokális j változóba. Az 5. sor kiíró utasításában látható, hogy a függvényben kezdetben i értéke megegyezik a globális i változó értékével (1), míg j értéke az átvett hívási paraméter értéke (3). A 6. sorban átírjuk j értékét: ez természetesen az f1 függvényben látható j nevű változó értékét írja át és nem befolyásolja sem a main függvény lokális j változójának, sem a globális j változónak az értékét. A 7. sor kiíró utasítása mutatja, hogy j értéke megváltozott. Az f1 függvény j értékét (7) adja vissza.

A main függvényben a 17. sor értékadó utasítása az i változónak értékül adja az f1 függvény visszatérési értékét (ami 3 volt). Tehát a main függvényben látható lokális i változó értéke 3 lesz, az itt látható j értéke pedig nem változott meg a függvényhívás alatt, ahogy az a 18. sor kiíró utasításában látható.

A blokk után a 20. sorban elhelyezett utasítás már nem látja a blokkban definiált változókat, így ezek már nem takarják el a main függvény lokális változóit: itt már ismét az eredeti i és j változókat láthatjuk.

A változók láthatósági tartományán kívül a változók élettartam is fontos jellemző. A globális változók a program futása alatt folyamatosan léteznek. A lokális változók akkor jönnek létre, amikor az őket definiáló függvény (vagy a változót definiáló blokk) meghívódik. A lokális változók megszűnnek, ha az őket definiáló függvény (vagy a változót definiáló blokk) véget ér.

9.7. példa: Vizsgáljuk meg a fenti példaprogram változóinak élettartamát.

A 2. sorban létrehozott i és j változók a program teljes futása alatt léteznek: tárhelyük már fordítási időben lefoglalódik és a program futása alatt ez nem változik meg. A függvények lokális változóinak a függvény meghívásakor foglalódik le tárhely: igaz ez mind az átvett paraméterekre (pl. a 4. sor j változója) és a lokálisan deklarált változókra is (pl. 12. sor változói). A lokális változók tárhelye megszűnik (értsd: később más célokra használódik fel, felülíró-

dik), amint a függvény befejezi működését. A blokkokban definiált változók tárhelye megszűnik, amint a blokk befejeződik, tehát a 15. sorban definiált i változó a 19. sornál megszűnik (bár a gyakorlatban a fordítók a változó tényleges megszűntetését gyakran csak a függvény befejezésekor végzik el – de erre természetesen nem szabad alapozni).

Megjegyzés: A programozási nyelvek általában lehetőséget adnak rá, hogy elkerüljük a lokális változók megszűntetését. Kérhetjük a fordítót arra, hogy lokális láthatóságú változóink értékét is őrizze meg, így a függvény következő hívásakor a korábban használt változó értéke újra elérhető. Erre a C nyelvben a statikus változók használhatók (pl. `static int i`).

9.3. Változók tárolása

Magas szintű programnyelvekben változókban tároljuk adatainkat. A változó egy szimbolikus név, amely a mögötte lévő adatra hivatkozik. A C nyelvben a változó neve a változó értékét jelenti (kivétel ez alól a tömb). A változók magas szintű absztrakciós lehetőséget biztosítanak, pl. egy hajó vagy kalóz adatait egyetlen változóban kezelhetjük.

A változókat általában a memóriában tároljuk (némely változók a processzor regisztereiben is helyet foglalhatnak, de ezzel most nem foglalkozunk). A memóriában tárolt változók helyét a memóriában a változó címe adja meg. A változó fontos tulajdonsága még a mérete: egy karakter tárolásához pl. 1 bájt, egy egész szám tárolásához 4 bájt, egy 100 elemű egész tömb tárolásához pedig 400 bájt memóriahely szükséges (az előbbi adatok azonban nyelvtől, platformtól és fordítótól függőek lehetnek).

9.8. példa: Egy programban egy x nevű egész, egy c1 és c2 nevű karakter, egy Q nevű duplapontos lebegőpontos szám, egy K nevű, 20 elemű karaktertömböt, egy karaktert és egész számot tartalmazó S nevű struktúrát, valamint egy T nevű lebegőpontos számot tárolunk. Vizsgáljuk meg, hogy milyen címeken tárolja programunk a változókat és a változóknak mekkora a méretük.

A C nyelvben a változók címét a & címoperátor adja vissza. A címeket a printf függvény segítségével a %p kapcsolóval írathatjuk ki hexadecimális formában.
A változók méretét a `sizeof` kulcsszóval kérdezhetjük le, ami a változó vagy típus méretét bájtokban adja meg.

Az alábbi program létrehozza a globális változókat és lekérdezi azok címét és méretét [*9.valtozok2.c*]:

```
#include <stdio.h>

int x=1;                                /* egész */
char c1='A';                            /* karakter*/
char c2='B';                            /* karakter*/
double Q=3.1415926;                     /* duplapontosságú lebegőp. */
char K[20]="Az ipafai papnak...";       /* karakter tömb */
struct vegyes{                          /* struktúra, benne */
   char C;                              /* karakter */
   int I;                               /* egész */
} S;
```

```
float T=1.5;                              /* lebegőpontos */

int main(){
   /* változók méretének és címének kiíratása */
   printf("x : meret=%2d, &x =0x%p \n", sizeof(x), &x);
   printf("c1: meret=%2d, &c1=0x%p \n", sizeof(c1), &c1);
   printf("c2: meret=%2d, &c2=0x%p \n", sizeof(c2), &c2);
   printf("Q : meret=%2d, &Q =0x%p \n", sizeof(Q), &Q);
   printf("K : meret=%2d, &K =0x%p \n", sizeof(K), K);
   printf("S : meret=%2d, &S =0x%p \n", sizeof(S), &S);
   printf("T : meret=%2d, &T =0x%p \n", sizeof(T), &T);
   return 0;
}
```

Figyeljük meg, hogy a változók címeit az & címoperátor segítségével kaptuk meg, kivéve a tömböt, ahol nem volt erre szükség, hiszen a tömb neve a C nyelvben a tömb címét jelenti. Egy PC-s környezetben Windows alatt gcc fordítóval a program a következő kimenetet adta:

```
x : meret= 4, &x =0x00402000
c1: meret= 1, &c1=0x00402004
c2: meret= 1, &c2=0x00402005
Q : meret= 8, &Q =0x00402008
K : meret=20, &K =0x00402010
S : meret= 8, &S =0x00404080
T : meret= 4, &T =0x00402024
```

Ebben a környezetben az int típus 4, a char 1, a double 8, míg a float 4 bájtos méretű volt. A tömb mérete megegyezik az alaptípus (char) méretének és a tömb hosszának szorzatával, jelen esetben ez 20 bájt volt, míg a struktúra méretét a fordító az alaptípusokból adódó 1+4=5 bájt helyett 8 bájtra kerekítette fel. A fordító program a fenti példában a változókat a memória 0x00402000 címétől kezdve helyezte el. Az első változó (x) mérete 4 bájt, ez után közvetlenül következik a két egybájtos karakter (c1 és c2). A c2 és Q között 2 bájt üres helyet találunk, majd kezdődik a 8 bájtos Q, közvetlenül utána a 20 bájt méretű K tömb, végül ez után a 4 bájtos T. Az S struktúrát a memóriában jóval feljebb helyezte el a fordító.

9.9. példa: Vizsgáljuk meg, hogyan tárolódnak a memóriában az egy bájtnál nagyobb méretű változók. A programunk tartalmazza az x nevű egész típusú változót és a K nevű, 20 elemű karaktertömböt [*9.valtozok3.c*].

Az alábbi program kiírja az x egész típusú változó értékét bájtonként, valamint a karaktertömb elemeit karakterként és hexadecimális számként is. Figyeljük meg az union használatát: a 4 bájtos egész szám mellé definiáltunk ugyanarra a memóriaterületre egy 4 elemű karaktertömböt. Mivel a karaktertömb elemei 1 bájtosak, ennek segítségével végig tudunk haladni a szám bájtjain és ki tudjuk azokat íratni. A tömb esetén szintén egy ciklussal végigmegyünk a tömbön és kiíratjuk a karaktereket, majd egy újabb ciklusban az ASCII karaktereket írjuk ki.

```
#include <stdio.h>

union teszt{
   int x;              /* Egy egész szám */
   unsigned char c[4]; /* u.a. a szám, de most bájtonként kezelve*/
} U;
unsigned char K[20] ="Az ipafai papnak...";   /* tömb */

int main(){
   U.x=0xC1CABAB1;              /* példa egész szám*/
   int i;
   printf("int x:\n");
   for (i=0; i<sizeof(U.x); i++){
      printf("%x ", U.c[i]);     /* egész szám kiírása bájtonként */
   }
   printf("\n\nchar K[20]:\n");
   for (i=0; i<sizeof(K); i++){
      printf("%c  ", K[i]);      /* karaktertömb kiír, karakterenként */
   }
   printf("\n");
   for (i=0; i<sizeof(K); i++){
      printf("%2x ", K[i]);      /* u.a. a tömb, most ASCII kóddal */
   }
   return 0;
}
```

A program kimenete a következő:

```
int x:
b1 ba ca c1

char K[20]:
A  z     i  p  a  f  a  i     p  a  p  n  a  k  .  .  .
41 7a 20 69 70 61 66 61 69 20 70 61 70 6e 61 6b 2e 2e 2e  0
```

Láthatjuk, hogy az egész számot (C1CABAB1) úgy tárolja a program, hogy a legkisebb helyiértékű bájt (B1) van a legalacsonyabb címen. Ezt a tárolási módot nevezzük Little Endian tárolásnak. Léteznek ún. Big Endian rendszerek is, ahol a legnagyobb helyiértékű bájt van a legalacsonyabb címen: egy ilyen rendszerben az `x` változót a fenti módon kiíratva a `c1 ca ba b1` karaktersorozatot kaptuk volna.

A futási eredményből jól látszik, hogy a tömbök tárolásakor a legalacsonyabb címen a tömb első (nulladik indexű) eleme foglal helyet, majd utána következnek sorban a tömb többi elemei. Ez független attól, hogy a rendszer Litte Endian vagy Big Endian tárolási módot használ.

Feladatok:

9.1. Módosítsuk a kalózok függvényeket használó adásvételi programját úgy, hogy a kiírás során a dátumok formátuma a következő példának megfelelő legyen:
Az Úr 1848. évének 3. havának 15. napján

9.2. Módosítsuk a fenti programot úgy, hogy a hónapokat nevükkel írja ki. pl.:
Az Úr 1848. évének március havának 15. napján

9.3. Módosítsuk a kalózok függvényeket használó adásvételi programját úgy, hogy tetszőleges fizetőeszközt lehessen az adásvételénél használni (6.5. feladat mintájára). Mely függvényeket kell módosítani? Mit kell ezen kívül a programban módosítani? Írjuk meg a C nyelvű programot.

9.4. Írjunk egy int CF(int x) és egy int FC(int x) függvényt, melyek átszámolják a bemeneti Celsius fokban megadott értéket Fahrenheit fokra (CF), illetve a Fahrenheit fokot Celsius fokra(FC). Írjunk programot, ami táblázatot készít vagy a Celsius-Fahrenheit, és a Fahrenheit-Celsius értékpárokról. A táblázatban elöl álló érték a -40 foktól +40 fokig terjedő intervallumban, egyesével változzon. A Celsiusról Fahrenheitre átváltó képlet a következő $X_F = 5/9(X_C - 32)$

9.5. Írjunk egy `int prim(int x)` függvényt, amely a bemeneti változóról eldönti, hogy prímszám-e: ha igen, akkor visszatérési értéke 1, különben pedig 0.
A `prim` függvény segítségével írjunk programot, amely egy megadott (bekért) számtartományban kiírja az összes prímszámot.

9.6. Készítsük el a 8.10. feladatban szereplő hisztogram-rajzoló programot úgy, hogy a hisztogramot egy `void hisztogram(int x[])` függvény rajzolja ki.

9.7. Egészítsük ki az előző programot úgy, hogy egymás után 10 véletlenszám-sorozatot generáljon, majd mind a 10 hisztogramot kirajzolja. Figyelem: a véletlenszám-generátor inicializálását csak egyszer kell elvégezni.

9.8. Egészítsük ki az előző programot úgy, hogy a számok generálását is egy `struct veletlenszamok generator(int N)` függvény végezze: a bemeneti paraméterként kapott számnak megfelelő számú véletlen számból álló sorozatot generáljon, majd ezt egy olyan struktúrában adja vissza, amelyben a sorozat egy MAXSZAM méretű (pl. 10000) tömbben szerepel, mellette pedig a sorozat mérete egy egész számban tárolódik.

9.9. Alakítsuk át a 9. fejezet 9.5. példájában írt `f2` függvényt úgy, hogy egyetlen tömböt és annak hosszát vegye át, majd ugyanebben a tömbben adja vissza az eredményeket.

9.10. Készítsünk programot, amelyben több függvényben is lokális változókat hozunk létre. Írassuk ki a lokális változók címét és méretét. Nézzük meg, mit tapasztalunk, ha vannak lokális és globális változóink is.

9.11. Készítsünk programot, amelyben egy bájtnál hosszabb elemekből, pl. egész számokból álló tömböt hozunk létre. Írassuk ki a tömb tartalmát bájtonként.

9.12. Készítsünk programot, amelyben egy struktúrákból álló tömböt hozunk létre. A struktúrában egy karakter és egy egész szám legyen. Írassuk ki a tömb tartalmát bájtonként.

10. fejezet

Összetett adatszerkezetek és manipuláló algoritmusaik

Adatszerkezeteink készítése közben a szekvenciális szerkezetek ábrázolására a struktúrákat, az iteratív adatszerkezetek használatára pedig a tömböket használtuk (ritkábban az unionok használatára is sor kerül szelekciót tartalmazó adatszerkezeteknél, de ezek használata nagyon hasonló a struktúrákéhoz).

A gyakorlatban előforduló – a valóságot jól modellező – adatszerkezetekben gyakran van szükség ezen adatszerkezetek bonyolultabb egymásba ágyazására. Ilyen bonyolultabb adatszerkezetekre néhány példa:

- Tömb, melynek elemei tömbök.
  - Pl.: matrix[1][2]
- Tömb, melynek elemei struktúrák
  - Pl.: hallgatok[3].nev
- Struktúra, melynek egyik mezője tömb
  - Pl.: varos.iskolak[2]
- Struktúra, melynek egyik mezője struktúra
  - Pl.: fejlesztoi_osztaly.vezeto.nev
- Tömb, melynek elemei tömböt (is) tartalmazó struktúrák
  - Pl.: varosok[3].iskolak[2]
- Struktúra, melyben struktúra tömb (is) van
  - Pl.:varos.iskolak[4].igazgato
- Tömb, melynek elemei struktúrát tartalmazó struktúrák
  - Pl.: fejlesztoi_osztalyok[5].vezeto.nev

10.1. példa: Írjuk fel reguláris definíciókkal egy városi iskolákat, óvodákat leíró adatszerkezetet.

Egy lehetséges megoldás, ahol az intézményekben előforduló személyek adatait tároljuk, a következő lehet:

Város → Iskolák_száma Iskolák Óvodák_száma Óvodák

Iskolák → Iskola*
Óvodák → Óvoda*
Iskola → Iskolaigazgató Tanárok_száma Tanárok Diákok_száma Diákok
Óvoda → Óvodaigazgató Óvónők_száma Óvónők Dadusok_száma Dadusok Gyerekek_száma Gyerekek
Tanárok → Tanár*
Óvónők → Óvónő*
Diákok → Diák*
Gyerekek → Gyerek*
Igazgató → Tanár
Óvodaigazgató → Óvónő
Tanár → Személy
Óvónő → Személy
Diák → Személy
Gyerek → Személy
Személy → Név Születési_hely

A fenti adatszerkezet ábrázolásakor a város egy struktúra lesz, amelynek mezői egy iskolákat tartalmazó és egy óvodákat tartalmazó tömb. Az iskola maga is egy struktúra, amelynek mezői az igazgató (egy személy típusú struktúra), egy tanárokat tartalmazó tömb (melynek elemei személy típusú struktúrák), egy diákokat tartalmazó tömb (melynek elemei szintén személy típusú struktúrák), valamint két egész, amely a tanárok és diákok számát tartalmazza. Hasonlóan az óvoda is egy struktúra, melynek mezői az igazgató (egy személy típusú struktúra), egy óvónőket tartalmazó tömb (melynek elemei személy típusú struktúrák), egy dadusokat tartalmazó tömb (melynek elemei személy típusú struktúrák), egy gyerekeket tartalmazó tömb (melynek elemei személy típusú struktúrák), valamint három egész, amely az óvónők, dadusok és gyerekek számát tartalmazza.

10.2. példa: Hozzuk létre a fenti adatszerkezeteket C nyelven.

Definiáljuk először a személyeket leíró struktúrákat. A nevek maximális hossza legyen MAXHOSSZ:

```
struct t_szemely{
   char nev[MAXHOSSZ];
   char szuletesi_hely[MAXHOSSZ];
};
```

Definiáljuk az iskolát és óvodát leíró struktúrát. Egy intézményben az egyszerűség kedvéért maximum MAXSZEMELY számú tanár, diák, óvónő, illetve gyerek járhat:

```
struct t_iskola{
   struct t_szemely igazgato;
   int tanarokszama;
   struct t_szemely tanar[MAXSZEMELY];
   int diakokszama;
   struct t_szemely diak[MAXSZEMELY];
};
```

```
struct t_ovoda{
   struct t_szemely igazgato;
   int ovonokszama;
   struct t_szemely ovono[MAXSZEMELY];
   int dadusokszama;
   struct t_szemely dadus[MAXSZEMELY];
   int gyerekekszama;
   struct t_szemely gyerek[MAXSZEMELY];
};
```

Definiáljuk most a város adatszerkezetét. Itt az intézmények maximális száma MAXINTEZMENY.

```
struct t_varos{
   int iskolakszama;
   struct t_iskola iskola[MAXINTEZMENY];
   int ovodakszama;
   struct t_ovoda ovoda[MAXINTEZMENY];
};
```

Végül hozzuk is létre egy város adatszerkezetét:

```
struct t_varos varos;
```

Kalóz Karcsi ismét módosította a kötélnyilvántartás követelményeit: a kötelek hosszán kívül a kötél felelősét is tárolni kell. A program először bekéri a kötelek számát, majd kötelenként bekéri a kötél hosszát és felelősét. A bevitel után a program bekér egy számot, majd kiírja az ennél hosszabb kötelek adatait.

10.3. példa: Készítsük el Kalóz Karcsi újabb, kötélfelelőst is nyilvántartó programját [*10.kotel1.c*].

A programot természetesen függvények segítségével oldjuk meg. Írunk egy függvényt, ami egy kötél adatait fogja bekérni (KotelBeker), ami egy kötél adatait írja ki (KotelKiir), ami az összes kötél adatát beolvassa (Beker), és ami megkeresi és kiíratja a kívánt tulajdonságú köteleket (Keres). Az adatszerkezet a következő lesz:

Kötéladatok → Kötelek_száma Kötelek

Kötelek → Kötél*

Kötél → Hossz Felelős

A program a következő:

```
#include <stdio.h>
#define ELEMSZAM 200
#define MAXNEV 20

struct t_kotel{              /* kötél adatok */
   double hossz;             /*   kötél hossza */
   char felelos[MAXNEV];     /*   kötél felelősének neve */
};

struct t_koteladatok{        /* összes kötél adatai */
   int  N;                              /* kötelek száma */
   struct t_kotel kotel[ELEMSZAM];      /* kötelek adatai */
};

/*
  Egy kötél adatainak beolvasása
  bemenet: standard input
  kimenet: visszatérési érték – kötél adatok
 */
struct t_kotel KotelBeker(){
   struct t_kotel kotel;
   printf("adja meg a kovetkezo kotel hosszat: ");
   scanf("%lf", &kotel.hossz);
   printf("adja meg a kotel felelosenek nevet: ");
   /* Ez csak egytagú nevekkel működik:*/
   scanf("%s", kotel.felelos);
   return kotel;
}

/*
  Egy kötél adatainak kiírása
  bemeneti paraméter: - kötél adatok
  kimenet: standard kimenet
 */
void KotelKiir(struct t_kotel kotel){
   printf("a kotel hossza: %lf, ", kotel.hossz);
   printf("a kotel felelose: %s \n", kotel.felelos);
}
```

```
/*
  Összes kötél adatainak beolvasása, a KotelBeker fv-t használja
  bemenet: standard input
  kimenet: visszatérési érték – kötél adatok
 */
struct t_koteladatok Beolvas(){
   struct t_kotel kotel;             /* egy kötél adatai */
   struct t_koteladatok kotelek;     /* kötelek adatai */
   int N;                            /* kötelek száma */
   int i;
   printf("Hany kotel van? ");       /* kötelek számának bekérése */
   scanf("%d", &N);
   kotelek.N=N;                      /* kötelek számának tárolása */
   for (i=0; i<N; i++){
      kotel=KotelBeker();            /* egy kötél adatainak bekérése */
      kotelek.kotel[i]=kotel;        /*         és tárolása */
   }
   return kotelek;
}

/*
  kötelek közül adott hossznál hosszabbak adatait kiírja
     KotelKiir fv-t használja
  bemeneti paraméter: kötelek adatai
  kimenet: standard kimenet
 */
void Keres(struct t_koteladatok kotelek){
   int ix;
   double x;
   printf("Hany meternel hosszabb kotelet keresunk? ");
   scanf("%lf", &x);
   for (ix=0; ix<kotelek.N; ix++){
      if (kotelek.kotel[ix].hossz > x)
         KotelKiir(kotelek.kotel[ix]);
   }
}

/*
  kötelek adatait bekéri és adott hossznál hosszabbak adatait kiírja
  bemeneti: standard input
  kimenet: standard kimenet
 */
```

```
int main(){
   struct t_koteladatok hajokotelek;
   hajokotelek=Beolvas();
   Keres(hajokotelek);
   return 0;
}
```

A fenti program sajnos csak olyan neveket tud tárolni, amelyben nem szerepel szóköz: a scanf függvény a szóközt határoló karakternek tekinti és a bevitelt megszakítja. Ha valódi neveket akarunk tárolni, akkor ilyen esetekben a teljes sort beolvasó fgets függvényt lehet használni. Ez a függvény visszaadja a bemenetén átadott karaktertömbbe a teljes beolvasott sort, szóközökkel együtt. A beolvasó függvényt a következőképpen kell módosítani:

```
struct t_kotel KotelBeker(){
   struct t_kotel kotel;
   printf("adja meg a kovetkezo kotel hosszat: ");
   scanf("%lf", &kotel.hossz);
   printf("adja meg a kotel felelosenek nevet: ");
   /* Ez csak egytagú nevekkel működik*/
   /* scanf("%s", kotel.felelos);   */
   /* előző scanf (kötélhossz) által bennhagyott karakterek kivétele:*/
   while (getchar()!='\n');
   fgets(kotel.felelos, MAXNEV, stdin);  /* teljes sor beolvasása */
   return kotel;
}
```

A fenti kódban az `fgets` függvényt használjuk a név beolvasására. Ez a megoldás azonban az előtte használt getchar() függvényhívás nélkül rosszul működne. Ugyanis a `scanf("%lf", &kotel.hossz)` függvényhívás a keresett szám (`%lf`) után megadott karaktereket a bemeneti pufferben hagyja. Ha soremeléssel zárjuk a szám bevitelét, legalább a soremelés karakter (\n) a pufferben marad. Ez azonban a következő `fgets` függvényhívást megzavarja: az `fgets` eddig a soremelésig fog olvasni, tehát a név bekérésekor azonnal egy üres sorral (üres névvel) tér vissza. Ha pedig a szám után véletlenül más karaktereket is megadtunk, akkor azokat fogja beolvasni. Ezért az `fgets` használata előtt ki kell üríteni a pufferben található karaktereket, pl. a példában látható `while (getchar()!='\n');` programsorral. A probléma kicsit bonyolultabb scanf formátumvezérlő használatával is megoldható lenne. Az alábbi `scanf` hívás a `while` ciklust és a `fgets` hívást is helyettesíti:

```
scanf("%*[^\n]%*[\n][^\n]", kotel.felelos);
```

A formátumvezérlő első kifejezése (`%*[^\n]`) a `while` ciklushoz hasonlóan eldob minden bejövő karaktert a \n-ig. A második formátumvezérlő kifejezés eldobja a \n karaktert, a harmadik pedig a sorvégéig beolvassa a kötél felelősének nevét. Azért nem ajánljuk mégsem ezt a megoldást, mert nem, vagy csak igen komplikáltan lenne megadható a MAXNEV, mint maximális névhossz. Ugyanis annak értékét (esetünkben 20-at) a 3. fomátumvezérlő % jele után kellene beírni, de az egy stringben van. Ezt csak trükkös makrókkal lehetne elérni, amivel itt nem foglalkozunk.

10.4. példa: Írjunk a kötélkezelő programhoz egy `KotelekKiir` nevű függvényt, ami egy `t_koteladatok` típusú struktúrában tárolt összes kötél adatát ki tudja írni [*10.kotel2.c*].

A függvény a következő lehet:

```
/*
  Összes kötél adatainak kiírása - KotelKiir fv-t használja
  bemeneti paraméter: - kötelek adatai
  kimenet: standard kimenet
 */
void KotelekKiir(struct t_koteladatok kotelek){
   int ix;
   for(ix=0; ix<kotelek.N; ix++){
         KotelKiir(kotelek.kotel[ix]);
   }
}
```

10.5. példa: Írjunk a kötélkezelő programhoz egy `Valogat` nevű függvényt, ami egy `t_koteladatok` típusú struktúrában tárol kötelek közül ki tudja válogatni azokat, amelyek hossza egy megadott tartományban van. Ezen kötelek adatait egy `t_koteladatok` típusú struktúrában adja vissza [*10.kotel2.c*].

A megoldás például a következő lehet:

```
/*
  kötelek közül adott hossz-tartományba esőket kiválogatja
  bemeneti paraméterek:
              kotelek: összes kötél adata
              min:     minimális kötélhossz
              max:     maximális kötélhossz
  kimenet: válogatott kötelek adatai
 */
struct t_koteladatok Valogat(struct t_koteladatok kotelek,
                             double min,
                             double max){
   struct t_koteladatok valogatas;
   int i;     /* a bemenő tömb indexe */
   int j=0;   /* a kimenő tömb indexe */
   for (i=0; i<kotelek.N; i++){
      if(kotelek.kotel[i].hossz > min &&
         kotelek.kotel[i].hossz<max){
         valogatas.kotel[j]=kotelek.kotel[i];
         j++;
      }
   }
   valogatas.N=j;
   return valogatas;
}
```

A válogatás során az `i` index segítségével végigmegyünk a bemenő `kotelek` nevű struktúra `kotel` mezőjének összes elemén és megvizsgáljuk, hogy az adott elem (`t_kotel` struktúra) `hossz` mezője teljesíti-e a követelményt. Amennyiben igen, az `i`-edik kötelet leíró struktúrát (`kotelek.kotel[i]`) átmásoljuk a `valogatas` nevű `t_koteladatok` struktúra `kotel` mezőjének j-edik pozíciójára (`valogatas.kotel[j]`, ahol `j` nulláról indult), majd növeljük `j`-t. Így `j` mindig a már leválogatott kötelek számát tartalmazza (és egyben a következő elem indexét a `valogatas.kotel` tömbben). Amint az összes elemet végignéztük, beállítjuk a `valogatas` struktúrában a kötelek számát (`valogatas.N=j`), majd visszatérünk a `valogatas` változó értékével.

A tesztelést és kipróbálást például a következő main függvénnyel végezhetjük el:

```
int main(){
   struct t_koteladatok hajokotelek;
   struct t_koteladatok valogatott_hajokotelek;
   hajokotelek=Beolvas();
   printf(" ***** Osszes kotel: \n");
   KotelekKiir(hajokotelek);
   valogatott_hajokotelek=Valogat(hajokotelek, 10.5, 20.7);
   printf(" ***** Valogatott kotelek: \n");
   KotelekKiir(valogatott_hajokotelek);
   return 0;
}
```

10.6. példa: Írjunk függvényt, ami tömbben tárolt egész számokat sorrendbe rendez. A függvény bemenete egy tömb, amelyben helyben elvégzi a rendezést és a kimenet a bemeneti – átrendezett – tömb lesz [*10.rendez.c*].

Számos rendezési algoritmus ismert, ezek közül most az egyik legegyszerűbbet, a minimumkiválasztásos rendezést valósítjuk meg. Az algoritmus a következő:

- Az első menetben válasszuk ki a teljes tömb legkisebb elemét, majd helyezzük ezt a tömb elejére (úgy, hogy megcseréljük a legkisebb elemet az első elemmel).
- A második menetben válasszuk ki a tömb legkisebb elemét a második elemtől kezdve, majd helyezzük ezt a tömb második helyére (szintén cserével: megcseréljük a legkisebb elemet az második elemmel).
- Ezt folytassuk annyiszor, ahány elemű a tömb. Az i-edik körben a minimumkeresést az i-edik elemtől (i-edik indextől) kell kezdeni és a legkisebb talált elemet ide kell tenni.

A függvénynek átadjuk a rendezni kívánt tömböt és a tömb méretét. A C nyelven megvalósított függvény pl. a következő lehet:

```
/*
  számokat hossz szerint rendez (helyben)
  bemeneti paraméterek:
              T:    számok tömbje
              N:    tömb hossza
  kimenet: a T tömbben keletkezik a rendezett számhalmaz
 */
void szamrendez(int T[], int N){
   int i, j;   /* segédváltozók */
   int minix;  /* legkisebb elem indexe */
   int csere;  /* segédváltozó a cseréhez */
   for (i=0; i<N; i++){/* az i. rendezett elemet keressük, elölről */
      minix=i;         /* a legkisebb, még nem rendezett elem indexe */
      /* min. keresése a még nem rendezett elemekben:*/
      for (j=i+1; j<N; j++){
         if (T[j]<T[minix])   /* ha kisebb elemet találtunk */
            minix=j;         /* megjegyezzük az indexét */
      }
      /* a következő i. elem a legkisebb még nem rendezett elem lesz */
      /* elemek cseréje: T[i] <--> T[minix]*/
      csere=T[i];
      T[i]=T[minix];
      T[minix]=csere;
   }
}
```

Kalóz Karcsi szeretné a programjába bevitt kötelek adatait úgy látni, hogy azok nagyság szerinti sorrendben legyenek rendezve.

10.7. példa: Írjuk át a számokat rendező függvényt úgy, hogy az alkalmas legyen a kötelek adatainak kezelésére [*10.kotel2.c*].

A kötél adatai között természetesen a hossz jellemző lesz a rendezési szempont, de a kötél ennél több adatot is tartalmaz (jelen esetben a felelős nevét is). Célszerű tehát a kötelet, mint egységet kezelni. Ahogy a számrendező számok tömbjét, úgy a kötélrendező kötelek tömbjét fogja rendezni:

```
void KotelRendez(struct t_kotel T[], int N){
   int i, j, minix;
   struct t_kotel csere;
   for (i=0; i<N; i++){
      minix=i;
      for (j=i+1; j<N; j++){  /* minimum keresése */
         if (T[j].hossz<T[minix].hossz)
            minix=j;
      }
```

```
      /* csere: T[i] <--> T[minix]*/
      csere=T[i];
      T[i]=T[minix];
      T[minix]=csere;
   }
}
```

A változásokat a fenti kódban pirossal jelöltük: természetesen a számok (`int`) helyett `t_kotel` típusú adatokkal dolgozunk, valamint az összehasonlításnál a kötél adatai közül a `hossz` mezőt kell használni.

A függvényt a kötélkezelő programból a következőképpen hívhatjuk meg a `hajokotelek` nevű struktúrában tárolt kötelekre [*10.kotel2.c*]:

```
KotelRendez(hajokotelek.kotel, hajokotelek.N);
```

Kalóz Karcsi szereti a sakkot és a számítógépeket is. Szeretne egy egyszerű sakkprogramot, de még egyik zsákmányban sem talált ilyent. Venni meg nem akar...

10.8. példa: Írjunk egy egyszerű programot, ami kirajzol egy sakktáblát, amire bábukat lehet elhelyezni. A bábukat a sakkban szokásos módon adjuk meg, tehát pl. ha a király B2 mezőn áll, akkor Kb2. Az egyszerűség kedvéért a világos bábukat nagy, a sötét bábukat kis betűkkel jelöljük [*10.sakktabla.c*].

A sakktábla egy mátrixnak fogható fel, amit programjainkban egy kétdimenziós tömbként kezelhetünk. A kétdimenziós tömb valójában egy olyan tömb, amelynek elemei tömbök. Hasonlóan az egydimenziós tömbhöz, a tömb elemeit indexeléssel érjük el, itt természetesen két index (oszlop és sor) segítségével. A tömb elemeire például a következő módon hivatkozhatunk:

```
T[1][3]=3;  A=T[2][1];
```

A tömb elemei természetesen bármilyen típusúak lehetnek. Példánkban az egyszerűség kedvéért minden bábu jelölésére használjunk egy karaktert (pl. k-király, h-huszár). A tábla minden mezőjét feleltessük meg a kétdimenziós tömb egy elemének: az A1 mező legyen a (0,0) indexű elem, az A2 mező a (0,1), ..., a B1 mező az (1,0), a B2 mező az (1,1), ..., a C3 mező a (2,2), ... stb. indexű legyen.

Ha a tábla adott mezőjén áll egy bábu, akkor a tömb megfelelő pozícióján tároljuk el a bábu jelölésére szolgáló karaktert. Tehát pl. a D2 mezőre egy világos királyt a következő módon helyezhetünk el: `Tabla[3][1] = 'K'`.

A C nyelvű program a következő lehet:

```
/*
 * Sakktábla rajzoló program
 */

#include <stdio.h>
#define MAXMERET 25


/*
  sakktábla méretének beolvasásaa
  bemenet: standard bemenet
  kimenet: visszatérési érték - tábla mérete
 */
int beker_N(){
int N;
printf("Tabla merete: ");
   scanf("%d", &N);
   return N;
}

/*
  sakktábla kirajzolása
  bemeneti paraméterek:
         tabla: sakktábla
         N: sakktábla mérete
  kimenet:
         standard kimenet
 */
void kirajzol(char tabla[MAXMERET][MAXMERET], int N){
   int i, j;
   char babu, szin;

   printf("  ");
   for (i=0;i<N; i++) printf("  %c", 'A'+i);    /* oszlopindexek*/
   printf("\n  +");
   for (i=0;i<N; i++) printf("--+", 'A'+i);     /* felső vonal */
   printf("\n");
   for(i=N-1;i>=0; i--){
      printf("%2d|", i+1);                     /* sorindex */
      for(j=0;j<N; j++){
         if ((i+j)%2)               /* világos vagy sötét mező? */
            szin=' ';
```

```
         else
            szin='X';
         if (babu=tabla[j][i])              /* ha van a mezőn bábu */
            printf("%c%c|", szin, babu);
         else                               /* ha nincs a mezőn bábu */
            printf("%c |", szin);
      };
      printf("\n  +");                          /* sor vége */
      for(j=0;j<N; j++) printf("--+", 'A'+j);    /* alsó vonal */
      printf("\n");
   }

}

/*
  Egy bábu pozíciójának beolvasása és eltárolása
  bemenet:
         paraméter: sakktábla
         standard inputról olvas
  kimenet:
         visszatérési érték: érvényes/nem érvényes pozíció
         bemenetként kapott sakktábla adatait módosítja
 */
int beker_pozicio(char tabla[MAXMERET][MAXMERET]){
   int oszlop_ix, sor_ix, ervenyes;
   char babu, inp[20];
   printf("Pozicio (pl. hA2): ");
   scanf("%s", inp);                        /* bemenet beolvasása */
   oszlop_ix=inp[1]-'A';                    /* oszlopindex számítása */
   sor_ix   =inp[2]-'1';                    /* sorindex számítása */
   babu     =inp[0];                        /* bábu azonosítója */
   if (ervenyes =(babu>='A' && babu<='z')) /* érvényes bemenő adat? */
      tabla[oszlop_ix][sor_ix]=babu;        /* tárolás */
   return ervenyes;                  /* érvényes flag visszaadása */
}
```

```
/*
  sakktábla inicilizálása
  bemeneti paraméter: sakktábla
  kimenet: inicializált sakktábla
 */
void init(char tabla[MAXMERET][MAXMERET]){
   int i,j;
   for (i=0; i<= MAXMERET; i++)
      for (j=0; j<= MAXMERET; j++)
         tabla[i][j]=0;  /* minden mezőre 0 */
}

/*
  bekéri és kirajzolja a sakktáblát
  bemenet: standard bemenet
  kimenet: standard kimenet
 */
int main(){
   char tabla[MAXMERET][MAXMERET];
   int N;
   N=beker_N();                    /* tábla méretének beolvasás */
   init(tabla);                    /* inicializálás */
   while(beker_pozicio(tabla));    /* babupozíciók beolvasása */
   kirajzol(tabla,N);              /* tábla kirajzolása */
   return 0;
}
```

A program `main` függvényében lefoglalunk egy MAXMERET-szer MAXMERET méretű kétdimenziós tömböt. Ebből csak akkora részt fogunk használni, amekkora a tábla méretének megfelel. Az `init` függvény null-karakterekkel tölti fel a táblát, ezzel jelezve, hogy nincs ott bábu. A `beker_N` függvény a tábla méretét olvassa be. A következő ciklus mindaddig fut, amíg a `beker_pozicio` érvényes adatot olvas be.

Figyelem: a `while(beker_pozicio(tabla));` ciklus magja üres, a ciklus csupán a feltételt értékeli ki mindaddig, míg az igaz. A feltétel kiértékelés mellékhatása a bábu beolvasása. A táblát a `kirajzol` függvény írja ki.

Figyeljük meg a `beker_pozicio` függvényben az index képzését: a bemenet egy karaktersorozat, melynek első eleme a bábu azonosítója, a második az oszlop-, a harmadik pedig a sor-azonosító (pl. kA5). Az oszlop-azonosító esetén az ’A’→0, ’B’ →1, C→2, stb. konverziót kell végrehajtani, amit legegyszerűbben úgy végezhetünk el, hogy az oszlopot jelző karakter kódjából kivonjuk az A karakter kódját. Sor esetén természetesen a sort jelző karakter kódjából ki kell vonni az 1 karakter kódját (nem 1-et!). A függvény akkor tekinti érvényesnek a bemenő adatot, ha a bábu azonosítóját jelző karakter betű. Így a bevitelt bármilyen nem betű karakterrel (pl. 0 vagy *) zárhatjuk.

A `kirajzol` függvény soronként rajzolja ki a táblát: ha van bábu az adott mezőn, akkor annak kódját is kiírja, valamint a mező színét is jelöli egy csillag karakterrel.

Egy példa futási eredmény a következő:

```
Tabla merete: 5
Pozicio: kA3
Pozicio: KD1
Pozicio: hB5
Pozicio: *
    A  B  C  D  E
  +--+--+--+--+--+
 5|* | h|* |  |* |
  +--+--+--+--+--+
 4|  |* |  |* |  |
  +--+--+--+--+--+
 3|*k|  |* |  |* |
  +--+--+--+--+--+
 2|  |* |  |* |  |
  +--+--+--+--+--+
 1|* |  |* | K|* |
  +--+--+--+--+--+
```

Amikor egydimenziós tömböket adtunk át függvényeknek, a tömb dimenzióját nem jelöltük a függvény deklarációjában, pl.:

```
void KotelRendez(struct t_kotel T[], int N)
```

A kétdimenziós tömbök esetén viszont a kódban azt látjuk, hogy a függvény deklarációja tartalmazza a tömb méretét is, pl.:

```
int beker_pozicio(char tabla[MAXMERET][MAXMERET])
```

Amikor statikus tömböket adunk át függvényeknek, a fordítónak tudnia kell, hogy mekkorák a tömb dimenziói. Pontosabban csak annyit kell tudnia, hogy az első dimenzión kívül a többi dimenzió mennyi. A tömbök tárolása ugyanis lineárisan történik: először letároljuk az első sort, majd a másodikat, stb. Az indexelés során egy sor és oszlopindexet adunk meg, ebből kell kiszámítani azt, hogy az indexelt elem hányadik a lineáris sorban.

Pl. Egy 2x3 méretű tömbben az elemek tárolása a következő sorrendben történik:

```
(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)
```

Egy 3x2 méretű tömbben pedig így:

```
(0,0), (0,1), (1,0), (1,2), (2,0), (2,1)
```

Ahhoz, hogy meg tudjuk határozni, hogy az (1,0) elem hol tárolódik (negyedik vagy harmadik pozíción), ismerni kell, hogy mekkora a tömb dimenziója. Pontosabban elég azt tudni, hogy milyen hosszúak a sorok (mekkora a második dimenzió), nem kell tudni azt, hogy összesen hány sor van (legalábbis akkor nem, ha nem akarjuk ellenőrizni az indexek helyességét). Ezért nem kell az egydimenziós tömbök esetén megmondanunk a tömb méretét és ezért kell a többdimenziós tömbök esetén megadni a dimenziók méretét a második dimenziótól kezdve. Így valójában elég lenne a következő függvénydeklaráció is:

```
int beker_pozicio(char tabla[][MAXMERET])
```

Feladatok:

10.1. Próbáljuk ki az `fgets` függvényt használó `KotelBeker` függvényt az `getchar()` függvényhívás nélkül. Először a megadott kötélhossz után soremelést üssünk. Ezután próbáljuk becsapni a programot: az adatbevitelnél ne zárjuk le a sort a kötél mérete után, hanem szóköz után adjuk meg a felelős nevét, majd ezután üssünk soremelést, pl. így: `adja meg a kovetkezo kotel hosszat: 57.6 Gyalog Fifi Bence¶` Miért működik most a program (eltekintve a kissé zavaros bemeneti képernyőtől)?

10.2. Tömbök rendezésének egyik kedvelt módszere a buborékos rendezés. Ez az algoritmus a tömb végéről indulva minden elemet összehasonlít az előző elemmel. Ha a két elem sorrendje nem megfelelő (tehát a nagyobb elem áll elöl), akkor megcseréli az elemeket. Így az első kör után a tömb első eleme garantáltan a legkisebb lesz. Ezt az eljárást folytatjuk, de most már a tömb végétől csak a második elemig haladunk, stb. N-1 kör után a tömb rendezett lesz. Készítsük el az algoritmus pszeudo-kódját. (A megoldás a következő feladatban látható)

10.3. A buborékos rendezés pszeudo-kódja a következő:

```
Eljárás Buborek
   Ciklus i=1-től (N-1)-ig
      Ciklus j=(N-1)-től i-ig -1-esével
         Ha Tömb(j-1)>Tömb(j) akkor
            Csere=Tömb(j-1)
            Tömb(j-1)=Tömb(j)
            Tömb(j):=Csere
         Elágazás vége
      Ciklus vége
   Ciklus vége
Eljárás vége
```

Programozzuk be az algoritmust C nyelven és teszteljük a működését.

10.4. Egészítsük a sakktábla-rajzoló programot úgy, hogy a bábuk bevitele után lehessen lépni is. A Lépéseket a kiinduló és cél mező azonosításával adjuk meg. Pl. az A3A3 lépés az A1 mezőn található bábut az A2 mezőre helyezi át. (A programnak nem kell ellenőrizni a lépések helyességét.) A program minden lépés után rajzolja ki a táblát. Ha * karaktert adunk meg lépésnek, a program fejezze be működését.

10.5. Egy sakktáblára pénzérméket helyezünk, minden mezőre kerülhet 0, 1, vagy akár több érme is. Az érméket úgy helyezzük el, hogy megadjuk az érme értékét és a mező pozícióját: pl. az 5B3 parancs egy 5 dukátos érmét helyez el a B3 mezőre. A bevitelt * karakter zárja. Ezután a program rajzolja ki a táblát, minden mezőn jelezve, hogy mennyi pénz van rajta (ahol nincs pénz, azt a mezőt hagyja üresen).

10.6. Egészítsük ki az előző feladatban írt programot úgy, hogy minden sor után kiírja az adott sorban lévő érmék összegét és minden oszlop alá kiírja az adott oszlopban lévő érmék összegét.

10.7. Tervezzük meg az adásvétel program kiterjesztett változatának adat- és programszerkezetét. A vételár lehessen több (maximum 10) tételből álló is: pl. 10 aranydukát, 1 papagáj és 1 rozsdás kard. A program mely függvényeit érinti a változás? Írjuk meg és teszteljük a programot C nyelven.

10.8. Tervezzük meg a kalóz cserebere programot, amely egy maximum 10 tételből álló tárgyhalmaz cseréjét egy másik, maximum 10 tételből álló tárgyhalmazra adminisztrálja. Az adásvételhez hasonlóan itt is kezelni kell a cserében részt vevő felek adatait és a csere dátumát. Valósítsuk meg és teszteljük a programot C nyelven.

10.9. Készítsünk a kalózok kötélkezelő programja számára menüt: legyen lehetőség (1) új kötél adatainak felvitelére, (2) a kötelek adatainak listázására, (3) a kötelek nagyság szerinti rendezésére, (4) adott hosszúságú kötelek listázására, valamint (0) a programból való kilépésre. Pl.

```
Nyomjá 1-est, ha ujj kötelet akarsz fővinni!
Nyomjá 2-est, ha listázni akarsz!!
Nyomjá 3-ast, ha naccság szerint akarsz rendezni!!!
Nyomjá 4-est, ha kötelet akarsz keresni hossza szerint!!!!
Nyomjá 0-t, ha ki akarsz lépni a programbul!
Ha mást nyomsz, lelőlek! Na, mit nyomsz?
```

A program minden parancs elvégzése után írja ki a menüt és várja a következő parancsot.

11. fejezet

A rekurzió és alkalmazása

A rekurzió alkalmazásával nagyobb problémák kisebb, hasonló problémák megoldására vezethetők vissza. Egy klasszikus példa a faktoriális számítása. Egy pozitív egész szám faktoriálisát pl. a következőképpen lehet definiálni:

$$n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot (n-1) \cdot n = \prod_{i=1}^{n} i$$

Ez úgynevezett iteratív megoldás. Ha egy programozási nyelvben ezt implementálni akarnánk, akkor egy ciklust alkalmaznánk, pl. így:

```
Függvény fact=EgyszeruFaktoriális(n)
   fakt=1
   i=1
   Ciklus amíg i<=n
      fakt=fakt*i
      i=i+1
   Ciklus vége
Eljárás vége
```

A faktoriális függvénynek létezik egy rekurzív definíciója is:

$$n! = \begin{cases} 1, & \text{ha } n = 1 \\ n \cdot (n-1)!, & \text{ha } n > 1 \end{cases}$$

Vagyis n faktoriális értékét úgy is kiszámíthatjuk, hogy $(n-1)$ faktoriálisát megszorozzuk n értékével. Itt a nagyobb probléma ($n!$) megoldását egy kisebb probléma ($(n-1)!$) megoldására vezettük vissza. A kisebb probléma is hasonló jellegű (faktoriális számítás), ezért a megoldás valóban rekurzív: a faktoriális számításához a faktoriális számítását használjuk fel.

Ha ezt egy programozási nyelven szeretnénk megfogalmazni, akkor pl. így járhatunk el:

```
Függvény fakt=Faktoriális(n)
   Ha n = 1 akkor
      fakt=1
   különben
      fakt=Faktoriális(n-1)*n
   elágazás vége
Eljárás vége
```

Jól látható, hogy a Faktoriális függvény saját magát hívja meg, de már eggyel kisebb bemenő paraméterrel. A meghívott függvény ismét meghívja önmagát, ismét kisebb eggyel kisebb paraméterrel, stb. A végtelen láncolat ott szakad meg, amikor a meghívott függvény bemenő paramétere 1-re csökken: ekkor nem történik újabb függvényhívás, hanem az utolsónak meghívott függvény visszaadja az 1 értéket. A rekurzív láncot megszakító kilépési feltételnek minden rekurzív függvényben léteznie kell, különben a függvény megállás nélkül hívná újra és újra önmagát. A függvények hívási lánca pl. a következőképpen alakul, ha a 4 faktoriálisát számítjuk ki (az egyszerűség kedvéért a Faktoriális függvénynév helyett az F rövidítést használtuk) :

```
F(4) = F(3) * 4                (F(4) elindul)
       F(3) = F(2) * 3         (F(3) is elindul)
              F(2) = F(1) * 2    (F(2) is elindul)
                     F(1) = 1    (F(1) is elindul majd visszatér)
              F(2) = 1 * 2     (F(2) visszatér)
       F(3) = 2 * 3            (F(3) visszatér)
F(4) = 6 * 4                   (F(4) visszatér)
```

A fenti hívási lánc összesen 4 mélységű, a lánc az `F(1)` függvény meghívásakor szakad meg: `F(1)` nem hívja tovább `F`-et, hanem visszatér; ezután rendre az `F(2)`, `F(3)` és `F(4)` függvényhívások is befejeződnek.

11.1. példa: Írjuk meg a rekurzív faktoriális függvényt C nyelven [*11.fakt.c*].

A függvény felépítése a fenti pszeudo-kóddal egyezik meg:

```
/*
  Faktoriális számítása: n!
  bemeneti paraméter: n
  visszatérési érték: n!
 */
int Faktorialis(int n){
   int fakt;
   if (n>1)
      fakt=Faktorialis(n-1)*n;     /* n!=n(n-1)! */
   else
      fakt=1;                      /* 1!=1 */
   return fakt;
}
```

Fontos megjegyezni, hogy a rekurzív függvények működésének fontos feltétele, hogy a függvény valamennyi meghívásakor a függvény egy új példánya induljon el a saját lokális változóival. Ilyen értelemben megtévesztő az a fogalmazás, hogy a függvény önmagát hívja meg: természetesen ugyanaz a programkód indul el, de minden meghívásakor saját változókészlete lesz.

A függvény első hívása legyen a `Faktorialis(4)`. Ennek a függvénynek vannak lokális változói és bemenő paraméterei (jelen esetben `fakt` és `n`), az ezeket a változókat tartalmazó környezetet nevezzük K_4-nek. Amikor ez a függvény ismét meghívja a `Faktorialis(3)` függvényt, akkor az új függvénypéldányban létrejön egy új, lokális környezet (K_3), amelyben ismét létezik egy `n` és `fakt` nevű változó; itt `n` értéke 3 lesz. Eközben a K_4 környezet a hívó függvényben továbbra is létezik (ahol az `n` változó értéke továbbra is 4). Ezután a `Faktorialis(2)` függvényhívással létrejön egy újabb(K_2) környezet, amelyben `n` értéke 2, miközben ezzel párhuzamosan a K_4 és K_3 környezetek továbbra is léteznek (ezekben `n`=4 és `n`=3). Végül a `Faktorialis(1)` függvényhívással létrejön egy negyedik (K_1) környezet is, ahol `n` értéke 1, de mellette még létezik az előző három is. Mikor a `Faktorialis(1)` hívás véget ér (a függvény visszatér az 1 értékkel az őt meghívó `Faktorialis(2)` függvénybe), a K_1 környezet megszűnik. Ezután a `Faktorialis(2)` tér vissza 2 értékkel, amikor is megszűnik a K_2 környezet, majd amikor `Faktorialis(3)` tér vissza 6 értékkel, akkor megszűnik K_3, majd végül, miután `Faktorialis(4)` visszaadta a 24 végeredményt, K_4 is megszűnik.

A hívási lánc legmélyebb pontján összesen 4 függvényhívás aktív, ezért négy példányban léteznek a függvények lokális változói és bemenő paraméterei (`n` és `fakt`). De ha pl. a 10000! értékére vagyunk kíváncsiak, akkor ezen változók 10000 példányban lesznek jelen. Ezért a rekurzív függvényhívások a bemenő paraméter (pontosabban a hívás mélységének) függvényében tetemes memóriát igényelhetnek.

Megjegyzések.

- A rekurzív módon való problémamegoldás némileg más gondolkodásmódot igényel, de ez gyakorlással elsajátítható és megtérül: a rekurzív függvények sok problémára nagyon frappáns, jól áttekinthető megfogalmazási lehetőséget adnak.
- Minden rekurzív algoritmus megfogalmazható iteratívan is (legfeljebb bonyolultabb módon).
- A iteratív megoldások általában gyorsabban futnak, mint a rekurzív algoritmusok.
- A függvényhívások paramétereiket és lokális változóikat a vermen tárolják. Minden újabb függvényhívás a verem tetejére teszi saját környezetét (majd miután a hívás véget ér, le is veszi onnan). Így a rekurzív függvényhívások a verem méretét növelik meg dinamikusan.

11.2. példa: A nyolc királynő problémája ismert logikai feladvány: helyezzünk el a sakktáblára 8 királynőt (vezért) úgy, hogy azok ne üssék egymást. Írjunk ezen probléma megoldására programot, mégpedig általánosan: egy N×N méretű táblán helyezzünk el N királynőt [*11.kiralyno.c*].

A feladatot a nyers erő módszerével egyszerűen meg lehet oldani: előállítjuk az összes $\binom{N^2}{N}$ darab lehetséges megoldást, majd ezeket leellenőrizzük, hogy kielégítik-e a szabályo-

kat. Ez azonban már elég kis N értékekre is használhatatlanul lassú lenne, pl. N=10 esetén már nagyságrendileg 10^{13} esetet kellene megvizsgálni. Felesleges azonban minden esetet végignézni, ennél ügyesebb algoritmus is adható. Tegyük fel a királynőket egyesével: ha a táblára már feltett királynők közül bármely kettő is üti egymást, akkor felesleges tovább próbálkozni: az eredmény mindenképpen rossz lesz. Készítsünk tehát egy olyan algoritmust, ami úgy próbálkozik - szisztematikusan – királynők lerakásával, hogy a rossz álláskezdeményeket azonnal észreveszi és az ez irányú próbálkozásokat abbahagyja. Az ilyen típusú algoritmusokat *backtracking*, vagy *visszalépéses kereső* algoritmusoknak nevezzük. A backtracking algoritmusokat gyakran rekurzív függvényekkel valósítjuk meg.

Az N-királynő problémára is nagyon szép rekurzív visszalépéses kereső megoldást adhatunk: Helyezzünk el az első sorban egy királynőt az első oszlopba. Ezután helyezzünk el a maradék N-1 sorban N-1 királynőt. Ha nem sikerült, akkor az első sorban a királynőt tegyük a következő mezőre. Ha így sem találunk megoldást, akkor tegyük arrébb ismét... A többi sorban hasonlóan járjunk el.

Az algoritmus megfogalmazása kicsit formálisabban:

```
Eljárás Felrak(Állás, sor)
   Ciklus oszlop=1-től N-ig
      Állás ← (sor, oszlop) pozícióra királynőt tesz
      Jó=Ellenőriz(Állás)
      Ha Jó = IGAZ akkor
         Ha sor = N akkor
            // megoldást találtunk!
            Kirajzol(Állás)
         Különben
            // eddig jó, folytassuk a következő sorral!
            Felrak(Állás, sor+1)
         Elágazás vége
      Elágazás vége
      // ezt kipróbáltuk, vegyük le
      Állás ← (sor, oszlop) pozícióról királynőt levesz
   Ciklus vége
Eljárás vége
```

Az állás a már felrakott királynőket tartalmazza, mégpedig az első sor–1 sorban. Az algoritmus ebből az állásból folytatja a próbálkozást, tehát a sor-adik sortól folytatja a felrakást. Egy ciklussal végigmegyünk az aktuális sor minden oszlopán és az aktuális oszlopba, tehát a (sor, oszlop) pozícióra leteszünk egy királynőt. Az Ellenőriz függvény dönti el, hogy a letett királynő szabályos helyen áll-e. Ha az aktuális állás szabályos és már az összes sorba tettünk királynőt, akkor egy helyes megoldást találtunk, ezt kiírjuk. Ha az aktuális állás szabályos, de még nem minden sorban van királynő, akkor folytatjuk a felrakást az aktuális állásból oly módon, hogy a Felrak függvényt meghívjuk az aktuális állásra, de a következő sorra. Miután

az aktuális oszloppozíciót kipróbáltuk, levesszük a királynőt és a következő oszloppozícióval próbálkozunk.

Amennyiben a függvényt az üres táblával az első sorra hívjuk meg (`Felrak(üres, 1)`), az megkeresi és kiírja az összes lehetséges megoldást.

Az állás reprezentációjáról nem szól a fenti pszeudo-kód. Lehetne egy NxN-es táblázatot használni, de jelen esetben elegendő, ha egy sokkal egyszerűbb adatszerkezettel írjuk le az állást: mivel minden sorban és oszlopban csak egy királynő állhat, így egyetlen vektorral is leírható az állás. Például a vektor i-ik eleme azt mondja meg, hogy az i-ik sorban melyik oszlopban áll királynő. Egy 4x4-es táblán pl. a [2 4 1 3] vektor azt jelenti, hogy az első sor 2. oszlopában (a B2 mezőn) áll egy királynő, a második sorban a 4. oszlopban (D2), a harmadik sor 1. oszlopában (A3), valamint a 4. sor 3. oszlopában (C4) áll királynő.

A program C nyelvű változatában ezt a leírást használtuk, de ott az indexek értelemszerűen 0-tól kezdődnek. A kód a pszeudo-kódban leírtakat követi, néhány kisebb változtatással:

- A feladat méretét a függvények átadják egymásnak (ezt lehetett volna pl. globális változóval is megoldani).
- A `felrak` függvény számolja az eddig talált megoldásokat. Ezt ebben a példában úgy oldottuk meg, hogy egy statikus változót (`szaml`) használunk. Ez ugyan láthatóságát tekintve lokális a `felrak` függvényben, de élettartamát tekintve az egész program futása alatt létezik (ellentétben a nem statikus lokális változókkal, amelyek megszűnnek a függvény befejezése után). Ez a változó hasonlóan viselkedik, mint egy globális változó (valójában a fordító is hasonlóan tárolja), de mégis csak a felrak függvényben lesz látható. A statikus változóból csak egyetlen példány létezik és az őt deklaráló függvény valamennyi meghívása ezt a példányt látja. Ez a változó csak kezdetben (a program indulásakor) inicializálódik (a program szerint 0 értékre), majd értékét két függvényívás között is megtartja. A `szaml` változó értékét a `felrak` függvény mindig növeli eggyel, amikor jó megoldást talál, így a `main` függvény ki tudja írni az összes megoldás számát.
- A `kirajzol` függvény a `szaml` értékét megvizsgálva csak az első 100 megoldást írja ki. Ezzel nagyobb méretek esetén a program futása jelentősen felgyorsul.

A programkód a következő [11.kiralyno.c]:

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

#define MAX 14

/*
  tábla kirajzolása
  bemenet:
     paraméterek:
     szamlalo: eddig talált megoldások száma
     n:        tábla mérete
     allas:    az aktuális állás.
  kimenet:
     standard kimenet
*/
void kirajzol(int szamlalo, int n, int allas[]){
   /* A program futásának gyorsítása érdekében csak az első
      száz megoldást rajzolja ki.
   */
   if (szamlalo>100) return;
   int i, j;
   printf("\n\n%4d. megoldas:\n  ", szamlalo);
   for (i=0; i<n; i++) printf(" %c", 'A'+i);
   printf("\n  +");
   for (i=0; i<n; i++) printf("-+");
   for (i=n; i>0; i--){
      printf("\n%2d|", i);
      for(j=0; j<n; j++) printf("%c|", allas[i-1]==j?'X':' ');
      printf("%d\n  +", i);
      for(j=0; j<n; j++) printf("-+");
   }
   printf("\n  ");
   for(i=0; i<n; i++) printf(" %c", 'A'+i);
}
```

```
/*
  állás ellenőrzése: nem ütik-e egymást a vezérek
  feltételezi, hogy az utolsó sor kivételével már ellenőrizve volt
  bemeneti paraméterek:
     allas:  az aktuális állás.
     sor:    az új sor, ebben kell elenőrizni az állást
  visszatérési érték: legális állás-e
*/
int ellenoriz(int allas[], int sor){
   int i;
   for(i=0; i<sor && allas[i]!=allas[sor] &&  /* ütés oszlopban? */
      abs(allas[i]-allas[sor])!=sor-i; i++);  /* ütés átlósan? */
   return (i==sor); /* ha i eléri sor-t, akkor nem volt ütés */
}

/*
  rekurzív felrakó függvény
  bemenet:
     paraméterek:
     meret: tábla mérete
     allas: aktuális felrakott állás.
            értelmezése: adott sor hányadik oszlopában van a vezér
     sor: ettől a sortól folytatódik a felrakás
  kimenet:
     az allas változó az új állást tartalmazza
     visszatérési érték: eddig talált megoldások száma

*/
int felrak(int meret, int allas[], int sor){
   int i, jo;
   /* A static kulcsszó miatt csak egy szaml lesz,
      és csak egyszer inicializálódik.
   */
   static int szaml=0;
   /* A sor minden oszloppozicióján végigmegyünk */
   for(allas[sor]=0; allas[sor]<meret; allas[sor]++){
      /* Addig megy, amíg a korábbi sorok vezérével nincs
         ütésben a vizsgált pozició.
      */
      jo=ellenoriz(allas, sor);
      /* Ha egyikkel sincs ütésben */
      if(jo){
         /* Ha nem az n. sorban vagyunk:
            felrak() fügvényhívás következő sorra (rekurzió).*/
         if(sor<meret-1)felrak(meret, allas, sor+1);
         /* Ha az n. sorba sikerült berakni,
```

```
            akkor egy megoldást találtunk! */
         else kirajzol(++szaml, meret, allas);
      }
   }
   /* A visszatérési érték az eddig talált megoldások száma. */
   /* Csak a main() függvényben használjuk fel. */
   return szaml;
}

int main(){
   int n, ret, sorallas[MAX];
   do{
      printf("A tabla merete [1-14]: ");
      ret=scanf("%d", &n);       /* táblaméret bekérése */
      while(getchar()!='\n');    /* puffer ürítése scanf után */
   }while(!ret || n<1 || n>MAX); /* amíg jó adatot nem adunk... */


   if(!(ret=felrak(n, sorallas, 0))) /* hívás a 0. sorra */
      printf("Az %dx%d meretu tablara nincs megoldas!\n", n, n);
   else if(ret>100)
      printf("\n\nOsszesen %d megoldas van!\n", ret);
   return 0;
}
```

Feladatok:

11.1. Írjuk át az N-királynő programot N-bástya programmá: helyezzünk el az N×N-es táblán N darab bástyát úgy, hogy azok ne üssék egymást.

11.2. Módosítsuk az N-királynő programot úgy, hogy első megoldás után álljon meg.

11.3. Módosítsuk az N-királynő programot úgy, hogy az állást egy N×N méretű tömbben tároljuk.

11.4. Írjunk rekurzív programot a Fibonacci-számok generálására. Az Fibonacci-számok definíciója a következő:

$$F_n = F_{n-1} + F_{n-2}, \text{ ha } n > 1$$
$$F_0 = 0$$
$$F_1 = 1$$

11.5. Írjunk rekurzív programot a Catalan-számok generálására. A Catalan-számok definíciója a következő:

$$C_n = \sum_{i=0}^{n-1} C_i C_{n-i-1}, \text{ ha } n > 0$$
$$C_0 = 1$$

(Az első néhány Catalan-szám a következő: 1, 1, 2, 5, 14, 42, 132, 429, 1430, 4862.)

12. fejezet

A programtervezés alapvető módozatai

Egyszerű programjaink készítése során nem okozott különösebb gondot, hogy a programok szerkezetét, a szükséges függvényeket, eljárásokat, adatszerkezeteket megtervezzük: ezek ilyen egyszerű esetekben természetesen „adódtak”, logikusnak tűntek. A feladat struktúrája természetes módon megjelent a program szerkezetében is, mind az adatszerkezetek, mind a vezérlési szerkezetek szintjén.

Bonyolultabb esetekben azonban a tervezési folyamatot célszerű tudatos módon, szisztematikusan végezni. A gyakorlatbak két alapvető megközelítést, gondolkodási módot találunk: az alulról fölfelé tervezést és a felülről lefelé tervezést.

A továbbiakban áttekintjük e két alapvető módszert. Azt azonban már most meg kell jegyezni, hogy sem egyik, sem másik módszerről nem lehet állítani, hogy jobb a másiknál: jelentős szemléletbeli különbség van a két megközelítés között. Így egyes gyakorlati esetekben hol egyik, hol a másik megközelítés bizonyulhat hatékonyabbnak. De egy valós mérnöki feladat megoldása közben valószínűleg egy kombinált módszert fogunk alkalmazni: a tervezés során váltogatni fogjuk a két módszert.

12.1. Felülről lefelé tervezés

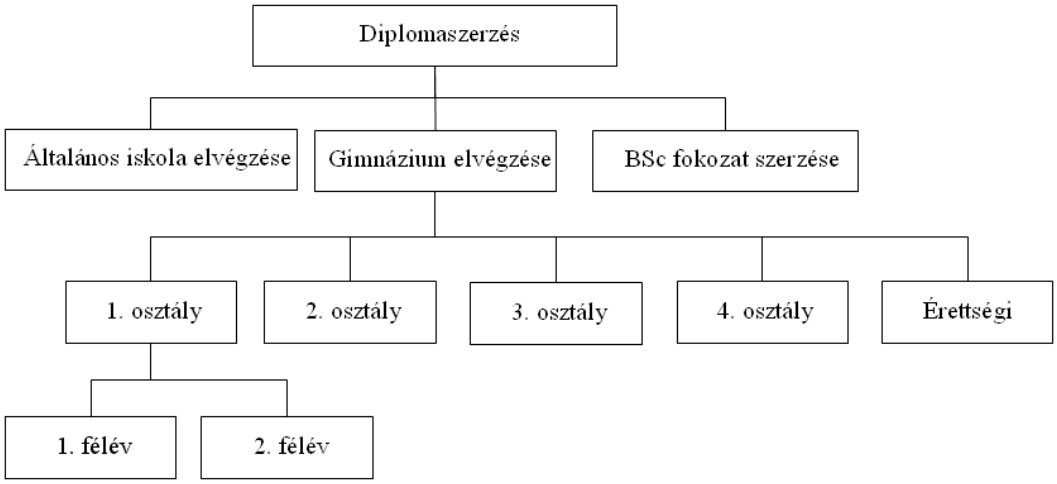
A felülről lefelé (top-down) tervezés alapelve az analízis: a célt elemezve azt kisebb logikai egységekre bonthatjuk, majd ezeket ismét tovább analizálva még kisebb logikai egységeket kapunk. Az analízis-részekre bontás folyamatot addig folytatjuk, míg már tovább nem bontható, elemi tevékenységekig jutunk.

Az felülről lefelé tervezés lépései a következők:

1. A feladatot (célt) egyetlen tevékenységként értelmezzük.
2. Ha van összetett tevékenységünk, akkor azt belátható számú, jól definiált feladatkörű tevékenységre bontjuk.
3. A tevékenységek sorrendjét az összetett tevékenységen belül meghatározzuk.
4. Ismétlés a 2. ponttól mindaddig, amíg csak elemi tevékenységek maradnak.

12.1. példa: Hogyan szerezzünk diplomát? (felülről lefelé tervezéssel)

Az első lépés szerint a diplomaszerzés tevékenységből, mint célból indulunk ki. Mivel ez nem elemi tevékenység, így a 2. lépés szerint ezt jól definiált feladatú részegységekre kell bontani. Egy ilyen logikus felbontás lehet, hogy el kell végezni az általános iskolát, a gimnáziumot, majd az egyetemen pl. BSc fokozatot kell szerezni. Ezzel az eredeti feladatot három kisebb feladatra dekomponáltuk. Ezen kisebb tevékenységek viszonya a feladatban egyszerű: a felsorolt sorrendben kell ezeket végrehajtani. Amennyiben ezen tevékenységek sem tekinthetők elemi tevékenységnek (figyelem: ez a végrehajtótól függ), akkor ezeket is tovább analizálhatjuk és bonthatjuk további részfeladatokra. Például a gimnázium elvégzése a négy év sikeres elvégzését és az érettségi letételét jelentheti. Az egyes tanévek további két félévre bonthatók, stb. A feladat végrehajtását Jackson-ábrán is leírhatjuk, ahogy az ábra mutatja. A felbontást tetszőleges finomságig folytathatjuk mindaddig, amíg a számunkra értelmezhető elemi szintig el nem jutunk. Hasonlóan az általános iskolai, valamint az egyetemi tevékenységek is tovább bonthatók.



12.1. ábra. A diploma megszerzésének folyamata felülről lefelé tervezéssel. Az ábrán még nincs minden tevékenység kifejtve.

Az felülről lefelé tervezés előnyei: a felülről lefelé tervezés szép megoldásokat eredményez, hiszen az adott feladathoz jól illeszkedő struktúrájú megoldást kapunk. A komplex feladattól egyre egyszerűbb részfeladatokon át jutunk el a megoldásig, minden szinten kezelhető méretű problémát oldunk meg. A közbülső lépések során definiált részfeladatok önálló szoftver-egységek (pl. függvények) lesznek, melyek elemi és később definiálandó összetett tevékenységeket tartalmaznak. A tervezés során definiált egységek kapcsolatai, interfészei jól definiáltak, ezért ezek önállóan is kezelhetők, párhuzamosan is fejleszthetők. A kódolás tehát könnyű és áttekinthető, az eredmény jól strukturált, logikus felépítésű lesz.

Az felülről lefelé tervezés hátrányai: A fejlesztés során felülről lefelé haladva egységeink definíciói olyan alegységekre hivatkoznak, melyek egyelőre csak funkcionálisan ismertek, ezek definíciója csak később készül el. Így egy magas szinten megtervezett és létrehozott egység nem tesztelhető – hiszen elemei még hiányoznak. Ezért a felülről lefelé tervezéssel készült szoftverek tesztelése csak a tervezési és fejlesztési folyamat legvégén kezdődhet el.

12.2. Alulról felfelé tervezés

Az alulról fölfelé tervezés (angol nyelvű szakirodalomból gyakran bottom-up tervezésnek is nevezik) alapelve a szintézis, már meglévő elemekből történő építkezés. Az alulról felfelé építkezés feltétele, hogy az elemi tevékenységeket ismerjük, ezeket kezdetben meghatározzuk. Ezen elemi tevékenységeket fogjuk logikusan rendezni, csoportosítani mindaddig, míg a tevékenységeknek egy megfelelő sorrendjét és hierarchiáját ki nem alakítjuk, amely hierarchia csúcsán a feladat megoldása áll.

Az alulról felfelé tervezés lépései a következők:

1. Az elemi tevékenységek meghatározása.
2. A tevékenységek függőségeinek feltárása.
3. A függőségek és a tevékenységek kapcsolódásainak figyelembe vételével a tevékenységek sorrendjének átrendezése úgy, hogy
   a. az új sorrend vegye figyelembe a függőségi viszonyokat és
   b. az új sorrendben a hasonló jellegű feladatok egymás mellé kerüljenek.
4. Az előbbi pontban hasonló jellegűnek ítélt tevékenységek egy összetett tevékenységbe zárása.
5. Ismétlés a 2. ponttól mindaddig, amíg egyetlen (összetett) tevékenység marad: ez a feladat megoldása

Az elemi tevékenységek meghatározása során nem feltétlenül a programnyelv elemi utasításaira, hanem a fejlesztői környezet által nyújtott, kész megoldásokra kell gondolni. Az elemi tevékenységek meghatározása gyakran az újrafelhasználás jegyében történik: amennyiben már van egy kész komponensünk, amely a feladathoz várhatóan jól használható, akkor azt használhatjuk elemi tevékenységként. Gyakori példa az alulról felfelé tervezés illusztrálására a LEGO játék: itt adott az elemkészlet, amelyből létre kell hozni a célunknak megfelelő játékot, például egy robotot. A robot végső kinézetét nagyban befolyásolja a rendelkezésre álló elemkészlet, de az építkezés során valószínűleg a robot lábát, törzsét, kezét, fejét állítjuk össze az elemi komponensekből, majd ezen elemeket állítjuk össze robottá.

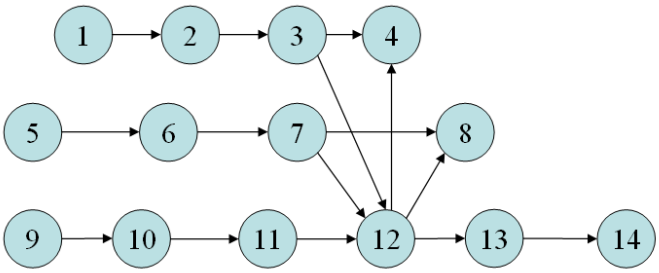
12.2. példa: Tervezzük meg a gyertyafényes vacsora menetét alulról felfelé tervezéssel.

Az első lépés szerint meghatározzuk, összegyűjtjük a felhasználható elemi tevékenységeket. Ez esetünkben pl. a következő lista lehet:

(1) gyertyát vásárlok
(2) a gyertyát az asztalra teszem
(3) a gyertyát meggyújtom
(4) a gyertyát eloltom
(5) szalvétát vásárolok
(6) a szalvétát összehajtom
(7) a szalvétát az asztalra teszem
(8) a szalvétát kidobom
(9) alapanyagot vásárolok
(10) vacsorát készítek

(11) tálalok
(12) eszünk
(13) leszedem a vacsorát
(14) elmosogatok

A második lépésben a tevékenységek függőségeit vizsgálom meg. Például a gyertya asztalra helyezése (2) célszerűen meg kell előzze a gyertya meggyújtását (3), valamint a gyertya eloltása (4) csak az evés (12) után következhet. Az ilyen típusú függőségek ábrázolásának eszköze a függőségi gráf lehet, amelynek csúcsai a tevékenységek, egy A és B csúcs között pedig akkor vezet irányított él (A-ból B-be mutató nyíl), ha B tevékenység függ A tevékenység végrehajtásától. Tehát a fenti probléma függőségi gráfjában pl. létezni fognak a (2) → (3) és (12) → (4) élek. A teljes függőségi gráf a 12.2. ábrán látható.



12.2. ábra. A gyertyafényes vacsora elemi tevékenységeinek függőségi gráfja

A harmadik lépésben a tevékenységek sorrendjét rendezzük át úgy, hogy a függőségi gráf kötöttségeit figyelembe vegyük. Formálisan: keressük az irányított függőségi gráf csúcsainak olyan $c_1, c_2, .., c_n$ sorrendjét, hogy bármely c_i és c_j-re igaz legyen, hogy ha a függőségi gráfba létezik c_i → c_j él, akkor $i < j$. Ráadásul egy olyan sorrendet próbálunk meg keresni, amelyben a hasonló jellegű feladatok egymás mellett foglalnak helyet. Ez a gyakorlatban a gráf egy olyan topológia rendezését jelenti, ahol a rendezett gráfban pl. csak balról jobbra vezetnek élek.

Példánkban a függőségi gráf csomópontjainak egy lehetséges rendezése a következő:

R1: (1), (2), (3), (5), (6), (7), (9), (10), (11), (12), (13), (4), (8), (14)

Egy másik lehetséges rendezés a következő lehet:

R2: (1), (5), (9), (2), (6), (10), (3), (7), (11), (12), (4), (8), (13), (14)

Ezen kívül a csúcsoknak (vagyis a tevékenységeknek) még számos lehetséges rendezése elképzelhető.

Ha az R1 rendezést használjuk, ahol olyan tevékenységeket helyeztünk egymás mellé, amelyeknek tárgya azonos (pl. a szalvéta, a gyertya): így is logikus felépítést kapunk, ha nem is célszerűt (ugyanis többször is el kell mennünk a boltba beszerezni a szükséges eszközöket). Az R2 rendezésben inkább a tevékenység hasonlósága szerint csoportosítottuk a tevékenységeket, így célszerűbb sorozatot kapunk. Az alulról fölfelé tervezés kihívása ennek a döntésnek a meghozása: milyen sorrendben, milyen csoportokat alkossunk, hogy programunk logikus, jól strukturált és hatékony legyen.

Az R2 rendezésből folytatva a tervezést a negyedik lépésben a logikailag összefüggő tevékenységeket egyetlen összetett tevékenységbe zárjuk. Pl. az (1), (5), (9) tevékenységek bevá-

sárlás jellegű tevékenységek, ezeket tekinthetjük egyetlen összetett tevékenységnek (Bevásárlás). A (2), (6), (10), (3), (7), (11) előkészületi tevékenységeket ismét egy közös tevékenységbe zárhatjuk (Előkészületek). A 12. tevékenység önállóan alkotja a Vacsora tevékenységet, míg a (4), (8), (13), (14) tevékenységek az Utómunkálatok összetett tevékenységbe zárhatók.

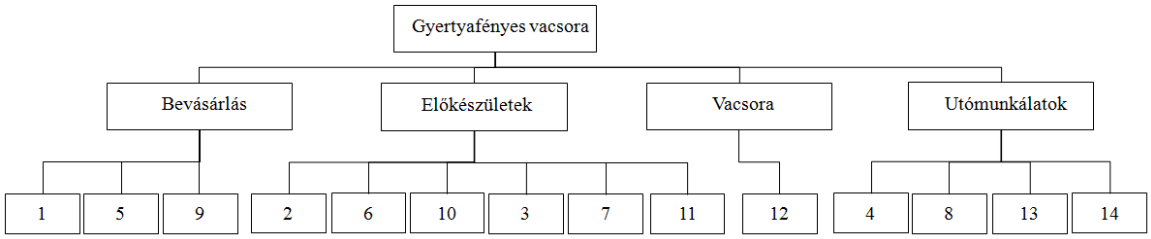
A tevékenységek összevonása után a következő (összetett) tevékenységeink adódtak:

(1') Bevásárlás = {(1), (5), (9)}
(2') Előkészületek = {(2), (6), (10), (3), (7), (11)}
(3') Vacsora = {(12)}
(4') Utómunkálatok = {(4), (8), (13), (14)}

Ezekre a tevékenységekre ismét folytatnunk kell a második lépéstől kezdve a folyamatot. Itt azonban már – a feladat egyszerűsége miatt – elérjük a végső célunkat: ezen négy tevékenység egymás utáni végrehajtása elvezet az eredeti feladat, a gyertyafényes vacsora végrehajtásához. Tehát a gyertyafényes vacsora a következő:

(1") Gyertyafényes vacsora = {(1'), (2'), (3'), (4')}

A teljes megtervezett folyamat a 12.3. ábrán látható.



12.3. ábra. A gyertyafényes vacsora folyamata

Az alulról felfelé tervezés előnyei: Az alulról felfelé tervezés nagyon jól támogatja a már létező szoftverkomponensek újrafelhasználását. A tervezés és implementálás már működő komponensekből történik, így a létrehozott összetett komponensek azonnal ki is próbálhatók, tesztelhetők. Mire a tervezési folyamat végére érünk, egy gyakorlatilag futtatható rendszer áll rendelkezésünkre.

Az alulról felfelé tervezés hátrányai: Mivel kész elemekből építkezünk, szükséges némi mérnöki intuíció ahhoz, hogy a létrehozandó komponenseket jól definiáljuk, mind az ellátandó feladatközöket, mind az interfészeket tekintve. Ezen komponensekből később nagyobb komponenseket építünk, így a komponensek utólagos egymáshoz illesztése komoly feladat lehet.

Feladatok:

12.1. Az alábbi elemi műveletekből alulról felfelé tervezéssel határozzuk meg a program struktúráját oly módon, hogy minden összetett művelet legfeljebb három másik műveletből álljon!

Bekér A, Bekér B, Bekér C, Kiír W, Kiír R, Kiír P, P=A*W+1, W=A+R+Q,R= négyzetgyök(Q*Q+Q), Q = négyzetgyök(A*A+B*B)

Rajzoljuk fel a program struktúráját Jackson-ábrával!

Valósítsuk meg a programot C nyelven, függvények segítségével, globális változók használatával.

Valósítsuk meg a programot C nyelven, függvények segítségével, globális változók nélkül.

12.2. Tervezzük meg a két szám legnagyobb közös osztóját kiszámító programot Euklideszi algoritmus segítségével, felülről lefelé tervezéssel.

Valósítsuk meg a programot C nyelven.

12.3. Írjuk le a felülről lefelé tervezés folyamatát pszeudo-kóddal, egy TopDown nevű rekurzív algoritmussal.

12.4. Írjuk le a felülről lefelé tervezés folyamatát pszeudo-kóddal, egy BottomUp nevű rekurzív algoritmussal.

13. fejezet

Irodalomjegyzék

[1] D. Bell, I. Morrey, J. Pugh, Programtervezés (Kiskapu, 2003)

[2] Herbert Schildt: C: The Complete Reference, 4th Ed. (McGraw-Hill Osborne Media, 2000)

[3] Benkő Tiborné, Benkő László, Tóth Bertalan: Programozzunk C nyelven! (ComputerBooks Kiadó, 1995)

[4] Brian W. Kernighan, Dennis M. Ritchie: A C programozási nyelv - Az ANSI szerint szabányosított változat (Műszaki Könyvkiadó, 1995)

[5] M.A Jackson: Problem Frames. (Addison-Wesley, 2001)

[6] Mérey András: Programtervezés Jackson-módszerrel (SZÁMALK, 1983)

[7] M. A. Jackson: Principles of Program Design. (Academic Press, 1975)

[8] N. Wirth: Algoriumisok + Adatstruktúrák = programok (Műszaki Könyvkiadó, 1982)

[9] N. Wirth, Program development by stepwise refinement, Comm. ACM, 14 (4), 1971.

[10] Andrew S. Tanenbaum: Számítógép-architektúrák (PANEM, 2006)

Elektronikusan elérhető kiegészítő szakmai anyagok:

[11] C szabványok: *http://www.open-std.org/jtc1/sc22/wg14/*

[12] Eredeti ASCII kód: *http://tools.ietf.org/html/rfc20*

[13] Egyéb karakterkódolások: *http://en.wikipedia.org/wiki/Character_encoding*

[14] Jackson-módszer: *http://en.wikipedia.org/wiki/Jackson_Structured_Programming*

[15] Struktúrált programozás: *http://en.wikipedia.org/wiki/Structured_programming*

[16] Scratch nyelv: *http://scratch.inf.elte.hu/*

[17] Bináris SI prefixek: *http://en.wikipedia.org/wiki/Binary_prefix*

F1. függelék.

ASCII kódolás

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	NUL	SOH	STX	ETX	EOT	ENQ	ACK	BEL	BS	HT	LF	VT	FF	CR	SO	SI
1	DLE	DC1	DC2	DC3	DC4	NAK	SYN	ETB	CAN	EM	SUB	ESC	FS	GS	RS	US
2		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
3	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
4	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
5	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
6	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
7	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

A táblázat használata: A karakter kódja hexadecimális formában a sor és az oszlop indexéből áll össze, ebben a sorrendben. Pl. az „A” karakter a 4. sorban és az első oszlopban található, így ASCII kódja a 0x41.

A nem nyomtatható karaktereket színes háttér jelöli.

F2. függelék.

A printf függvény legfontosabb formátumvezérlői

```
#include <stdio.h>

int main(){
   printf("Egy printf-fel tobb sornyi\ninformaciot is ki lehet
iratni.\nTetszoleges helyen sort\nemelhetunk \"\\n\"
ujsor\nkarakterrel.\n\n");
   //Egy printf-fel tobb sornyi
   //informaciot is ki lehet iratni.
   //Tetszoleges helyen sort
   //emelhetunk "\n" ujsor
   //karakterrel.
   //
   printf("Tudunk specialis karaktereket is kiirni:\nmacskakorom:
\"\nvisszaper: \\\nszazalekjel: %%\n\n");
   //Tudunk specialis karaktereket is kiirni:
   //macskakorom: "
   //visszaper: \
   //szazalekjel: %
   //
   printf("A \"\\t\" tabulator
hasznalataval\nnyolc\tkarakterenkent\ttabulalhatjuk\ta\tkiirando\tinformaci
ot.\n\n");
   //A "\t" tabulator hasznalataval
   //nyolc   karakterenkent  tabulalhatjuk   a       kiirando
informaciot.
   //
   printf("\t\t\t\t\"\\t\\t\\t\\t\\t\" hatasara itt kezdem a
kiirast,\r\"\\r\"-t hasznalva itt folytatom.\n\n");
   //"\r"-t hasznalva itt folytatom. "\t\t\t\t" hatasara itt kezdem a
kiirast,
   //
   printf("Decimalis elojeles egesz kiirasa \"%%d\" format string-gel
tortenik, pl.: %d\n\n", -123);
```

```
   //Decimalis elojeles egesz kiirasa "%d" format string-gel tortenik, pl.:
-123
   //
   printf("Irjuk ki 1-tol 5-ig az egesz szamokat, parameterkent,
tabulatorral:\n%d\t%d\t%d\t%d\t%d\n\n", 1, 2, 3, 4, 5);
   //Irjuk ki 1-tol 5-ig az egesz szamokat, parameterkent, tabulatorral:
   //1       2       3       4       5
   //
   printf("Irjuk ki ugyanezt, a [width] mezot
hasznalva:\n%8d%8d%8d%8d%8d\n\n", 1, 2, 3, 4, 5);
   //Irjuk ki ugyanezt, a [width] mezot hasznalva:
   //       1       2       3       4       5
   //
   printf("Irjuk ki ugyanezt balra igazitva, [flag] es [width] mezoket
hasznalva:\n%-8d%-8d%-8d%-8d%-8d\n\n", 1, 2, 3, 4, 5);
   //Irjuk ki ugyanezt balra igazitva, [flag] es [width] mezoket hasznalva:
   //1       2       3       4       5
   //
   printf("Hasznaljuk a [.precision] mezot is, hogy az alabbi output
legyen:\n%-8.4d%-8.4d%-8.4d%-8.4d%-8.4d\n\n", 1, 2, 3, 4, 5);
   //Hasznaljuk a [.precision] mezot is, hogy az alabbi output legyen:
   //0001    0002    0003    0004    0005
   //
   printf("Irjuk ki ugyanezt nullakkal balrol feltoltve, [flag] es [width]
mezoket hasznalva:\n%08d%08d%08d%08d%08d\n\n", 1, 2, 3, 4, 5);
   //Irjuk ki ugyanezt nullakkal balrol feltoltve, [flag] es [width]
mezoket hasznalva:
   //00000001000000020000000300000004000000005
   //
   printf("Irjuk ki az egesz szamokat egymas ala a sor elejere -2-tol 2-
ig:\n%d\n%d\n%d\n%d\n%d\n\n",-2, -1, 0, 1, 2);
   //Irjuk ki az egesz szamokat egymas ala a sor elejere -2-tol 2-ig:
   //-2
   //-1
   //0
   //1
   //2
   //
   printf("Hasznaljuk a [flag] mezot, hogy a szamok egymas ala keruljenek
elojeltol fuggetlenul:\n% d\n% d\n% d\n% d\n% d\n\n",-2, -1, 0, 1, 2);
   //Hasznaljuk a [flag] mezot, hogy a szamok egymas ala keruljenek
elojeltol fuggetlenul:
   //-2
   //-1
   // 0
   // 1
   // 2
   //
   printf("Hasznaljuk a [flag] mezot, hogy a szamok elojele mindig
megjelenjen:\n%+d\n%+d\n%+d\n%+d\n%+d\n\n",-2, -1, 0, 1, 2);
   //Hasznaljuk a [flag] mezot, hogy a szamok elojele mindig megjelenjen:
   //-2
```

```
   //-1
   //+0
   //+1
   //+2
   //
   printf("Tobb [flag]: az elobbieket 8 karakteren, elojellel, balrol
nullaval kiegeszitve:\n%+08d\n%+08d\n%+08d\n%+08d\n%+08d\n\n",-2, -1, 0, 1,
2);
   //Tobb [flag]: az elobbieket 8 karakteren, elojellel, balrol nullaval
kiegeszitve:
   //-0000002
   //-0000001
   //+0000000
   //+0000001
   //+0000002
   //
   printf("Tobb [flag]: 8 karakteren, + elojel helye kihagyva, balrol
nullaval kiegeszitve:\n% 08d\n% 08d\n% 08d\n% 08d\n% 08d\n\n",-2, -1, 0, 1,
2);
   //Tobb [flag]: 8 karakteren, + elojel helye kihagyva, balrol nullaval
kiegeszitve:
   //-0000002
   //-0000001
   // 0000000
   // 0000001
   // 0000002
   //
   printf("Irjuk ki a 12-ot parametereket hasznalva, az alabbi
szamrendszerekben:\ndecimalis %d = oktalis %o = hexadecimalis %x\n\n", 12,
12, 12);
   //Irjuk ki a 12-ot parametereket hasznalva, az alabbi szamrendszerekben:
   //decimalis 12 = oktalis 14 = hexadecimalis c
   //
   printf("Irjuk ki ugyanezt, hasznaljuk a [flag] mezot a szamrendszerek
jelolesere:\n%d = %#o = %#x\n\n", 12, 12, 12);
   //Irjuk ki ugyanezt, hasznaljuk a [flag] mezot a szamrendszerek
jelolesere:
   //12 = 014 = 0xc
   //
   printf("Irjuk ki ugyanezt, hexadecimalis szamoknal a 10-15 szamjegyek A-
F legyen:\n%d = %#o = %#X\n\n", 12, 12, 12);
   //Irjuk ki ugyanezt, hexadecimalis szamoknal a 10-15 szamjegyek A-F
legyen:
   //12 = 014 = 0XC
   //
   printf("Igy nez ki a Pi 2 (3.14), 4 (3.1415) es 10 (3.1415926535)
tizedesjegy pontossaggal megadva \"%%e-vel\":\n%e %e %e \n\n", 3.14,
3.1415, 3.1415926535);
   //Igy nez ki a Pi 2 (3.14), 4 (3.1415) es 10 (3.1415926535) tizedesjegy
pontossaggal megadva "%e-vel":
   //3.140000e+000 3.141500e+000 3.141593e+000
   //
```

```
   printf("Igy nez ki a Pi \"%%E-vel\":\n%E %E %E \n\n", 3.14, 3.1415,
3.1415926535);
   //Igy nez ki a Pi "%E-vel":
   //3.140000E+000 3.141500E+000 3.141593E+000
   //
   printf("Igy nez ki a Pi \"%%f-el\":\n%f %f %f \n\n", 3.14, 3.1415,
3.1415926535);
   //Igy nez ki a Pi "%f-el":
   //3.140000 3.141500 3.141593
   //
   printf("Igy nez ki a Pi \"%%g-vel\":\n%g %g %g \n\n", 3.14, 3.1415,
3.1415926535);
   //Igy nez ki a Pi "%g-vel":
   //3.14 3.1415 3.14159
   //
   printf("Igy nez ki a Pi \"%%G-vel\":\n%G %G %G \n\n", 3.14, 3.1415,
3.1415926535);
   //Igy nez ki a Pi "%G-vel":
   //3.14 3.1415 3.14159
   //
   printf("Probaljuk ki hogy viselkedik a \"%%g\" nagy szamok eseten,
peldaul a 10^8*Pi-vel:\n%g \nMint lathatjuk, ilyenkor exponencialis alakot
hasznal.\n\n", 314159265.35);
   //Probaljuk ki hogy viselkedik a "%g" nagy szamok eseten, peldaul a
10^8*Pi-vel:
   //3.14159e+008
   //Mint lathatjuk, ilyenkor exponencialis alakot hasznal.
   //
   printf("A [width] hasznalataval szebben jelenithetjuk meg a
szamainkat:\n%12f %12f %12f \n\n", 3.14, 3.1415, 3.1415926535);
   //A [width] hasznalataval szebben jelenithetjuk meg a szamainkat:
   //    3.140000     3.141500     3.141593
   //
   printf("A [.precision] hasznalataval tizedesjegyeket
veszthetunk:\n%12.3f %12.3f %12.3f \n\n", 3.14, 3.1415, 3.1415926535);
   //A [.precision] hasznalataval tizedesjegyeket veszthetunk:
   //       3.140        3.142        3.142
   //
   printf("A [.precision] viszont ujabb tizedesjegyeket is
biztosithat:\n%12.10f %12.10f %12.10f \n\n", 3.14, 3.1415, 3.1415926535);
   //A [.precision] viszont ujabb tizedesjegyeket is biztosithat:
   //3.1400000000 3.1415000000 3.1415926535
   //
    printf("%10s(Az \"almafa\" szo 10 karakter szelessegben kiirva.)\n\n",
"almafa");
   //    almafa(Az "almafa" szo 10 karakter szelessegben kiirva.)
   //
   printf("%-10s(Az \"almafa\" szo 10 karakter szelessegben, balra
igazitva.)\n\n", "almafa");
   //almafa    (Az "almafa" szo 10 karakter szelessegben, balra igazitva.)
   //
```

```
   printf("%10.3s(Az \"almafa\" szo elso harom karaktere 10 karakter
szelessegben.)\n\n", "almafa");
   //       alm(Az "almafa" szo elso harom karaktere 10 karakter
szelessegben.)
   //
   printf("%-10.3s(Az \"almafa\" szo elso harom karaktere 10 karakter
szelessegben, balra igazitva.)\n\n", "almafa");
   //alm       (Az "almafa" szo elso harom karaktere 10 karakter
szelessegben, balra igazitva.)
   //
   return 0;
}
```

F3. függelék.

A C nyelv operátorai és ezek precedenciái

Operátor	Precedencia	Leírás	Asszociativi-	Példa

()	1	függvényhívás, csoportosítás (részkifejezések precedenciájának növelése)	⇨	`int i=-3, j;`
				`j=abs(i);`
				`// i:-3; j:3`
				
				`j=3*(4+5);`
				`// j:27`

[]		tömb indexelés	⇨	`int tomb[3]={1, 12, 23}, elem;`
				`elem=tomb[1];`
				`// elem:12`

.		struktúra tagjának kiválasztása	⇨	`struct datum{`
				`  unsigned int Evszam;`
				`  unsigned int Honap;`
				`  unsigned int Nap;`
				`};`
				
				`struct szemely{`
				`  char Nev[21];`
				`  struct datum Szuletesi_ido;`
				`}Fiu;`
				`…`
				`  Fiu.Szuletesi_ido.Evszam=1990;`
				`  Fiu.Szuletesi_ido.Honap=11;`
				`  Fiu.Szuletesi_ido.Nap=29;`

->		struktúra tagjának kiválasztása a struktúrára mutató pointer alapján	⇨	`struct datum{`
				`  unsigned int Evszam;`
				`  unsigned int Honap;`
				`  unsigned int Nap;`
				`};`
				
				`struct szemely{`
				`  char Nev[21];`
				`  struct datum Szuletesi_ido;`
				`}Fiu;`
				
				…
				`  struct szemely *pFiu=&Fiu;`
				`  pFiu->Szuletesi_ido.Evszam=1990;`
				`  pFiu->Szuletesi_ido.Honap=11;`
				`  pFiu->Szuletesi_ido.Nap=29;`

++		post-inkremens	⇨	`int i=2, j;`
				`j=i++;`
				`// i:3; j:2`

--		post-dekremens	⇨	`int i=2, j;`
				`j=i--;`
				`// i:1; j:2`

++	2	pre-inkremens	⇦	`int i=2, j;`
				`j=++i;`
				`// i:3; j:3`

--		pre-dekremens	⇦	`int i=2, j;`
				`j=--i;`
				`// i:1; j:1`

+		plusz előjel (nincs hatása)	⇦	`int i=2, j;`
				`j=+i;`
				`// i:2; j:2`

-		negatív előjel	⇦	`int i=2, j;`
				`j=-i;`
				`// i:2; j:-2`

!		logikai tagadás	⇦	`int i;`
				`i=!(2<3);`
				`// i:0 (hamis)`

~		bitenkénti negálás	⇦	`char c1=0x0f, c2;`
				`c2=~c1;`
				`// c1:0b00001111; c2:0b11110000`

(type)		kifejezés típusának megváltoztatása (Cast)	⇦	`int i=2, j=4;`
				`double a, b;`
				`a=i/j;`
				`b=(double)i/j;`
				`//a:0.0; b:0.5`

*		indirekció (objektumra hivatkozás az objektum címe alapján)	⇦	`struct datum{`
				`  unsigned int Evszam;`
				`  unsigned int Honap;`
				`  unsigned int Nap;`
				`}Datum;`
				
				`…`
				`  struct datum *pDatum=&Datum;`
				`  (*pDatum).Evszam=1990;`

&		cím (objektum memóriacíme)	⇦	`int i, *pi;` `pi=&i;`

sizeof		típus vagy objektum mérete byte-ban	⇦	`struct datum{`
				`  unsigned int Evszam;`
				`  unsigned int Honap;`
				`  unsigned int Nap;`
				`}Datum;`
				
				`…`
				`  int i=sizeof(int*);`
				`  int j=sizeof(Datum);`
				`  //i:4 (32 bites architektúra)`
				`  //j:12`

*	3	szorzás	⇨	`double a=3.2, b=4.1, c;`
				`c=a*b*2.0;`
				`//c:26.24`

/		osztás	⇨	`int i=3, j=2;`
				`double a, b;`
				`a=i/j; //egész értékű osztás`
				`b=a/j; //lebegőpontos osztás`
				`//a:1.0; b:0.5`

%		modulus	⇨	`int i=7, j=4, k;`
				`double a=7.0, b=4.0;`
				`k=i%j;`
				`//k=a%b;// fordítási hibát okoz`
				`//(modulus csak egész típusokon`
				`//értelmezett)`
				`//k:3 (a 7/4 maradéka 3)`

+	4	összeadás	⇨	`double a=3.2, b=4.1, c;`
				`c=a+b;`
				`//c:7.3`

-		kivonás	⇨	`double a=3.2, b=4.1, c;`
				`c=a-b;`
				`//c:-0.9`

<<	5	bitenkénti balratolás	⇨	`int i=15, j;`
				`j=i<<4;// i értéke NEM változik!`
				`//i:15; j:240; (szorzás 2^4=16-tal)`

>>		bitenkénti jobbratolás	⇨	`int i=15, j;`
				`j=i>>3;// i értéke NEM változik!`
				`//i:15; j:1; (egész osztás 2^3=8-cal)`

<	6	kisebb reláció	⇨	`int i=2<3;`
				`//i:1 (igaz)`
				`int a=2, b=4, c=3;`
				`// Az alábbi egy hibás vizsgálat arra,`
				`// hogy b értéke a és c közé esik-e:`
				`i=a<b<c;`
				`//i:1 (igaz)`
				`// értelmezés: a<b -> 1 (igaz)`
				`// 1<c -> 1 (igaz)`

<=		kisebb vagy egyenlő reláció	⇨	
>		nagyobb reláció	⇨	
>=		nagyobb vagy egyenlő reláció	⇨	

==	7	egyenlő reláció	⇨	`int i=1==2>0;`
				`//i:1 precedencia miatt a`
				`// kiértékelési sorrend: 1==(2>0)`

!=		nem egyenlő reláció	⇨	

&	8	bitenkénti és (AND)	⇨	`int i, flags=0x63;//0b01100011`
				`int mask=0x0f;  //0b00001111`
				`i=flags&mask;  //i:0b00000011`
				`// maszkolás: csak a maszkban`
				`// szereplő bitekre vagyunk kíváncsiak`

^	9	bitenkénti kizáró vagy (XOR)	⇨	`char c1='a';  //0b01100001`
				`char c2='B';  //0b01000010`
				`char mask=0x20;//0b00100000`
				`c1=c1^mask;  //0b01000001`
				`c2=c2^mask;  //0b01100010`
				`//c1:'A'; c2:'b'`
				`//a maszkban szereplő bit(ek)et negál-juk`

|	10	bitenkénti vagy (OR)	⇨	`char c1='a';  //0b01100001`
				`char c2='B';  //0b01000010`
				`char mask=0x20;//0b00100000`
				`c1=c1|mask;  //0b01100001`
				`c2=c2|mask;  //0b01100010`
				`//c1:'a'; c2:'b'`
				`//a maszk bit(ek)et 1-re állítjuk`

&&	11	logikai és (AND)	⇨	`int i, a=2, b=4, c=3;`
				`// Egy helyes vizsgálat arra,`
				`// hogy b értéke a és c közé esik-e:`
				`i=a<b&&b<c; //precedencia:`
				`(a<b)&&(b<c)`
				`//i:0 (hamis)`

||	12	logikai vagy (OR)	⇨	`char i, c1='A', c2='s', c3='Z';`
				`i=c2<c1||c2>c3;`
				`//i:1 (igaz, c2 nem nagybetű)`

?:	13	feltételes (három operanduszú)	⇦	`int i, j=-2;`
				`i=j<0?-j:j;`
				`// i:2 (i=abs(j))`

=	14	értékadás	⇦	`int i=1, j=2, k=3, l=4;`
				`// többszörös értékadás`
				`i=j=k=l;`
				`// i:4; j:4; k:4; l:4`

+=, -=, *=, /=, %=, &=, ^=, |=, <<=, >>=		összetett érték-adás	⇦	`int i=3;`
				`i*=4+5;`
				`//i:27`
				`//precedencia: i=i*(4+5)`

,	15	felsorolás	⇨	`int i, j=1, k=2, l=3, m=4;`
				`i=(j, k, l, m);`
				`//i:4`
				`//a felsorolás utolsó kifejezése szá-mít`

F4. függelék.

Az elektronikus melléklet tartalma

[6.heron.c]: Háromszög területének meghatározása Héron képlete alapján.
[6.haromszog_s.c]: Háromszög kerületének és területének meghatározása, struktúrák használatával
[6.adasvetel1.c]: Hajó adásvétel program struktúrák használatával
[7.haromszoge.c]: Eldönti, hogy 3 szám jelentheti-e egy háromszög oldalait
[7.papagaj1.c]: Hajó és papagáj adásvétel program
[7.papagaj2.c]: Hajó és papagáj adásvétel program, unionnal
[8.kotel1.c]: Kötélnyilvántartó program, első verzió
[8.kotel2.c]: Kötélnyilvántartó program, második verzió
[8.kotelmax.c]: Kötélnyilvántartó program, leghosszabb kötél meghatározása
[8.karaktermasol1.c]: Karakterenkénti másolás bemenetről kimenetre
[8.karaktermasol2.c]: Karakterenkénti másolás bemenetről kimenetre, elöltesztelő ciklussal
[8.karakterszamol.c]: Karakterek számolása és osztályozása szövegben
[8.szamkitalal.c]: „Gondoltam egy számot” játék
[9.haromszog_fv.c]: Háromszög kerületének és területének meghatározása függvények használatával
[9.adasvetel2_fv.c]: Hajó adásvétel program függvények használatával
[9.tombfv1.c]: Tömb kezelése függvényben, visszaadás struktúrában
[9.tombfv2.c]: Tömb kezelése függvényben, visszaadás bemenő paraméterben
[9.valtozok1.c]: Globális és lokális változók, függvényparaméterek viselkedése
[9.valtozok2.c]: Változók címe és mérete
[9.valtozok3.c]: Tömbök és nagyobb méretű változók tárolása (endiannes)
[10.kotel1.c]: Kötélnyilvántartó, felelősökkel
[10.kotel2.c]: Kötélnyilvántartó, felelősökkel, rendezéssel, stb.
[10.rendez.c]: Számok sorba rendezése minimumkiválasztásos rendezéssel
[10.sakktabla.c]: Sakktábla rajzolása
[11.fakt.c]: Faktoriális függvény, rekurzióval
[11.kiralyno.c]: N-királynő problémája, rekurzív visszalépéses kereséssel
[F2.printf.c]: A printf függvény legfontosabb formátumvezérlő parancsai