Papp Róbert

Programozás (elmélet)

Jegyzet

A műszaki középiskolák második osztálya (mechatronikai technikus és számítógép elektrotechnikus szakokon), valamint első osztálya (információs technológiák elektrotechnikusa szakon) számára

Előszó

Ez a jegyzet egy fontos űrt próbál betölteni. Ugyanis a Vajdaság területén élő magyarság – habár él az anyanyelven való tanulás lehetőségével – rengeteg problémába ütközik. Ezek közül az egyik problémát az anyanyelven elérhető tanulási eszközök, a tankönyvek hiánya okozza. Az általános iskolákban, valamint a gimnáziumokban még jó a helyzet, a szakközépiskolákban viszont annál rosszabb. Habár a legtöbb általános tantárgyból már létezik magyar nyelven tankönyv, az igazi problémát a szaktantárgyak okozzák. Nagyon sok esetben még szerb nyelven sem létezik megfelelő tankönyv, a magyar nyelven létező tankönyvek száma pedig minimális.

Ami a műszaki szakközépiskolák Programozás nevű szaktantárgyát illeti, szerb nyelven sok esetben már létezik tankönyv. Ilyen például a számítógép elektrotechnikus szakra írt tankönyv második osztályra. Habár a tantárgy megy tovább harmadik és negyedik évben is, a második osztály számára írt tankönyvnek van talán a legnagyobb jelentősége, mivel itt sajátítják el a tanulók a C-programozás alapjait, amit mintegy belépőkártyaként visznek tovább a későbbi évekbe. Ezért a másodikos Programozás tantárgy magyar nyelven elérhető tankönyve nagy jelentőséggel bírna. Már azért is, mert más szakokon is létezik ez a tantárgy, így például mechatronikai technikus szakon (második év), valamint az új információs technológiák elektrotechnikusa szakon (első év). Habár vannak kisebb eltérések, ami a tantervet illeti, a tananyag szinte ugyanaz. Mivel a mechatronikai technikus szak Programozás tantárgya (második év) tananyagból egy kicsit többet tartalmaz (a függvényeket is), ezért ez az anyagrész is be lett téve a jegyzetbe (más szakokon a függvényeket egy évvel később tanulják). Így, habár ez a jegyzet az előbb említett szakokra lett optimalizálva, más szakok tanulói is felhasználhatják, amennyiben C-programozást kell tanulniuk, és természetesen egyéb diákok is, akik saját akaratukból szeretnének megismerkedni a programozás világával.

Maga a jegyzet a mechatronikai technikus szak második éves Programozás tantárgy tantervét követi. Ami a felhasznált szakirodalmat illeti, természetesen a megfelelő második osztály számára írt szerb nyelvű tankönyv lett alapul véve, kivéve a függvények részt, amihez fel lett használva a harmadik osztály számára írt szerb nyelvű tankönyv is. Magyarországi szakirodalom is fel lett használva a magyar szakkifejezések és alternatív magyarázási módszerek miatt. Azonban a tankönyvek és szakkönyvek bonyolult és talán a részletekbe túlságosan belemenő írásmódja miatt inkább célul az lett kitűzve, hogy a jegyzet érthető legyen egy középiskolás számára is. Ezért a szerző arra törekedett, hogy saját szavaival magyarázza el a fontos dolgokat, és inkább az alapokat magyarázza el részletesen és többféleképp. Ez a felhasznált szakirodalomra nem jellemző, mivel ott a szerzők feltételezik, hogy az olvasó már rendelkezik egy alapvető tudással, amit középiskolákban nem szabad feltételezni. A felhasznált szakirodalom megtalálható a jegyzet végén.

A jegyzet négy fő fejezetre van osztva. Az első a programozás fogalmát próbálja megmagyarázni, és úgy lehet rá tekinteni, mint egy bevezetőre. A második fejezet továbbra sem fog igazi programozási fogalmakat bemutatni, itt az algoritmusokon van a hangsúly, amik programnyelvektől függetlenek, és az algoritmusos gondolkodásmód elsajátítása a cél, ami sok kezdőnek okoz fejfájást. A harmadik fejezet lényegében az előzőnek a folytatása, itt gyakorlati szemszögből lesznek az algoritmusok bemutatva, méghozzá vizuálisan – folyamatábrák segítségével. A jegyzet legfontosabb és egyben legterjedelmesebb fejezete a negyedik fejezet, ami a C programozási nyelvet mutatja be, kezdve az előfeldolgozóval, azonosítókkal és változókkal, folytatva az elágazós és ciklusos struktúrákkal, és befejezve a tömbökkel és függvényekkel.

Az olvasók többsége észre fogja venni, hogy pár alcímben szerepel a „haladó szint” kifejezés. A szerző többéves tapasztalata szerint vannak a tantervnek olyan részei, amik vagy túl bonyolultak egy átlagos középiskolás számára, vagy annyira kevésszer vannak a későbbiekben

használva, hogy talán nem érdemes velük annyit foglalkozni. Ezek a részek azonban nem lettek teljesen kivágva, hanem inkább a teljesség kedvéért a „haladó szint” utótaggal lettek ellátva. Azok az olvasók, akik valóban szeretnének mélyebb tudásra szert tenni, ajánlatos ezeknek az alcímeknek az elolvasása is, azonban a jegyzet úgy lett írva, hogy azoknak az olvasóknak se legyen hiányérzetük, akik úgy döntenek, hogy kihagyják ezeket a részeket. Még egyszer ki kell hangsúlyozni, hogy ezek a részek nem feltétlenül a nehézségük, vagy bonyolultságuk miatt lettek áttéve a haladó szintre, hanem azért, mert nincs nagy gyakorlati jelentőségük a későbbi tantervben.

A végén még ki kell hangsúlyozni, hogy ez a jegyzet csak a Programozás tantárgy elméleti részét tartalmazza. A gyakorlati rész, ami ugyanolyan fontos, mint az elméleti, feladatokat tartalmaz magyarázatokkal, úgy is felfogható, mint egy példatár. Szükség esetén tervben lenne a példatár elkészítése is jegyzet formájában.

Papp Róbert, MSc

2015. VI. 30.

Topolya, Szabadka

Tartalomjegyzék

1. fejezet: Bevezetés a programozásba

Ebben a bevezető fejezetben röviden a programozásról, mint aktivitásról lesz szó. Először azt fogjuk definiálni, hogy mi az a számítógépes program, mit jelent a programozás, mint aktivitás, és kik azok a programozók. Ezután a programozási nyelveket és a három fő kategóriájukat fogjuk kicsit bővebben bemutatni. Természetesen beszélni kell az integrált fejlesztői környezetről is, amik lényegében olyan segédprogramok, amik pont a programozást könnyítik meg. Ezután rátérünk a szintaxis és a szemantika definiálására. A fejezetet a szintaxisdiagramokkal zárjuk.

A programról és a programozásról

Mielőtt belemerülnénk a programozás világába, illik előtte pár szót szólni arról, mi is az adat és az adatok feldolgozása, amik szervesen kapcsolódnak a számítógépekhez és magához a programozáshoz.

Mint ismeretes, az **adat** (ang. ***Data***) lehet valamiféle tény, fogalom vagy esemény. Az emberek már a történelem korai fázisában rájöttek arra, hogy az állandó fejlődés szemszögéből kulcsfontosságú az adatok lejegyzése, tárolása. Az emberek hamar rájöttek arra is, hogy a meglévő adatokból új adatokat vagy információkat nyerhetnek, így került előtérbe az adatok feldolgozása.

Def: az adatokon való műveletek rendszeres végrehajtását **adatfeldolgozásnak** (ang. *Data Processing*) nevezzük. □

Az adatfeldolgozás által a meglévő adatok állandóan változnak, alakulnak, új adatok születnek. Az adatfeldolgozás célja az, hogy a meglévő (sokszor jelentéktelen) adatokból olyan új adatokra, akár információkra vagy tudásra tehessünk szert, amikre nem jöttünk volna rá feldolgozás nélkül.

Az adatfeldolgozás alapjában véve négy fázisból áll:

- **Bemenetel** – az adatok begyűjtését, bevitelét jelenti. Ezeken az adatokon akarjuk végrehajtani a feldolgozást.
- **Feldolgozás** – célja a bemeneti adatok transzformációja, azaz feldolgozása.
- **Kimenetel** – a feldolgozott adatokból új adatok születnek. Hívhatjuk őket akár eredménynek is.
- **Tárolás** – ha úgy érezzük, hogy az újonnan kapott adatok fontosak, akkor azokat tárolni kell, annak érdekében, hogy később akár ismét feldolgozhassuk őket.

Habár az emberek a számítógépeket elsősorban arra találták fel, hogy segítsenek a számolásban, hamar rájöttek, hogy a számítógépek legnagyobb erénye az adatfeldolgozás. Erre szükség is volt, hiszen az emberiség fejlődésével a begyűjthető adatok mennyisége is rohamosan nőni kezdett. Nem szabad elfelejteni, hogy alapjában véve a számolás is az adatfeldolgozás egy formája – számokon történő műveletek végzése. Azonban az adatfeldolgozás magában foglalja szöveges dokumentumok, képek, diagramok, hangok, stb. feldolgozását. A mai számítógépek erre mind képesek.

Ezzel el is érkeztünk a program és a programozás fogalmához. Ugyanis a számítógép alapjában véve egy olyan gép vagy készülék, ami alkalmas arra, hogy a bevitt adatokon egyszerű

műveleteket hajtson végre. Ezeket a műveleteket hívjuk **gépi műveleteknek**. A gépi műveletek pár tipikus példája: aritmetikai műveletek (összeadás, kivonás, szorzás, osztás), logikai műveletek (kisebb, nagyobb, egyenlő, nem egyenlő), adatok beolvasása a bemenetről, adatok kiadása a kimenetre, stb.

Azonban nem sokat érnek a gépi műveletek, ha nincsenek adatok, amiken azokat végre lehet hajtani. Vegyük például az összeadás műveletet. Tételezzük fel, hogy ennek a műveletnek a gépi szimbolikus megfelelője az `ADD` kifejezés[1]. Az `ADD` gépi művelet nem sokat ér, ha nem adunk neki két adatot (ebben az esetben két számot), amivel dolgozhat. Azonban ha hozzárendeljük a két hiányzó számot, vagy változót (hivatalos néven *operandust*) és társítunk hozzá még egy harmadikat, amihez lesz hozzárendelve az eredmény, a művelet értelmet nyer:

```
2 ADD 3 = x
```

Def: azt a gépi műveletet, amihez hozzátársítottuk a szükséges operandusokat, ami által a művelet értelmet nyert, **gépi utasításnak** vagy **gépi parancsnak** nevezzük. □

Def: Ahhoz, hogy megoldjunk egy problémát, legtöbbször több gépi utasítást is végre kell hajtani, méghozzá meghatározott sorrendben. A gépi utasítások meghatározott sorrendben megadott sorozatát **gépi programnak**, vagy csak egyszerűen **programnak** nevezzük. □

Egy számítógép sem tud működni program nélkül. Már a számítógép felkapcsolásakor végrehajtódik egy alapprogram, aminek feladata a komputer belső alkatrészeinek, tehát elemeinek a felismerése és beállítása. Az operációs rendszer és az azon futó alkalmazások is programok. A játékok, vírusok, mind programok. Még az internetes honlapok beolvasása is lehetetlen programok nélkül. A nyomtató, hangkártya, stb. nem működne eszközillesztő program (ang. *Driver*) nélkül. Mindezeket a programokat el is kell készíteni, és ezt a műveletet hívjuk **programozásnak**, azokat a személyeket, akik pedig programok írásával foglalkoznak, **programozóknak**.

A programozás egy *kreatív* tevékenység, és alapjában véve három fázisból áll:

1. **Tervezés** – ki kell találni, milyen utasításokat kell majd végrehajtani a programban ahhoz, hogy megoldjuk vele az adott problémát, azaz létre kell hozni a megoldás *algoritmusát*.[2]
2. **Program írása** – az előző fázisban felvázolt ötletet „el kell magyarázni” a számítógépnek. Ez valamilyen *programnyelvben* történik – máskülönben a számítógép nem fogja megérteni, mit akarunk általa végrehajtani.
3. **Tesztelés** – a program megírása után a programot tesztelni kell a számítógépen, ugyanis nem biztos, hogy a megírt program helyes eredményt hoz vissza. Ha hibára lelünk, azt javítani kell.

A programnyelvek

Mint már említettük, ahhoz, hogy egy problémát sikeresen megoldjunk a számítógépen, azt be kell a komputernek mutatni, el kell neki „magyarázni”. Mint tudjuk, az emberek közti kommunikáció valamilyen **nyelv** által történik. A nyelv nem más, mint nyelvi jelek (betűk, vagy hangok) és szabályok (nyelvtan) összessége. Az emberek egymás között *természetes nyelven* kommunikálnak, viszont egy ember kommunikálhat géppel is, sőt, két gép is kommunikálhat

1 Az angol „*add*” szó magyar megfelelője „*hozzáadni*”.

2 Az algoritmusokról bőven lesz szó a következő fejezetben.

egymással. Azonban egy számítógéppel nem kommunikálhatunk természetes nyelven. Annak érdekében, hogy ez a kommunikáció mégis létrejöjjön, különféle *mesterséges nyelvek* jöttek létre. Ezeket hívjuk programnyelveknek.

Def: a **programnyelv** (vagy **programozási nyelv**) egy olyan az ember által olvasható és értelmezhető nyelv, amivel közvetlenül vagy közvetve kommunikálhatunk egy számítógéppel, feladatokat adva neki. □

Fel lehet tenni a kérdést: miért jöttek létre a programnyelvek? A válasz igen egyszerű. Mint ismeretes, a számítógépek ún. **gépi nyelven** működnek, ami alapjában véve a **bináris számrendszeren** alapszik. Tehát a gépi nyelv egy két „betűből", nullákból és egyesekből, álló nyelv. Ez előnyös a komputereknek, de teljesen idegen az embereknek. Annak érdekében, hogy minél egyszerűbben tanulható és elsajátítható nyelven kommunikáljunk a számítógéppel, a mérnökök létrehozták a programnyelveket. Ezek direkt rokonságban állnak a gépi nyelvvel, de a nyelvezetük inkább a természetes nyelvekhez hasonlít. Egy programnyelvre úgy is tekinthetünk, mint egy arany középútra a gépi és a természetes nyelv között.

A programnyelveket három fő kategóriába sorolhatjuk:

1. **Gépi nyelv** – a számítógép direkt bináris nyelve („`0`" és „`1`"). Habár a számítógépek feltalálásával az első mérnökök gépi nyelven kommunikáltak a számítógéppel, ez a módszer hamar le lett váltva.
2. ***Assembly*** – lényegében ugyanaz, mint a gépi nyelv, csak a bináris ábécé lett leváltva az angol ábécére, illetve a gépi műveleteknek könnyebben megjegyezhető angol kifejezéseket rendeltek (pl. így jött létre az `ADD` művelet), aminek köszönve sokkal érthetőbb lett az emberek számára. Az *Assembly*-t hívják még *„szimbolikus gépi nyelvnek"*, vagy *„alacsony szintű nyelvnek"* is.
3. **Magas szintű nyelvek** – még „emberibb" grammatikával rendelkeznek, mivel hiányoznak belőlük az *Assembly* kissé furcsa és nyers szabályai. Manapság a programozók többsége magas szintű nyelven programozik. Ma nagyon sok magas szintű programnyelv létezik. Ebből csak párat sorolunk fel: *Algol*, *FORTRAN*, *COBOL*, *BASIC*, *Pascal*, *Modula-2*, *Ada*, *LISP*, *Prolog*, *C*, *C++*, *C#*, *Delphi*, *Java*, stb.

Az integrált fejlesztői környezet

Minden programot, amit megírtuk papíron, vagy valamilyen szövegszerkesztő programban, még nem tudjuk rögtön lefuttatni, ugyanis ehhez szükség van arra, hogy a programot *előkészítsük*. Erre a célra lettek kitalálva az integrált fejlesztői környezetek.

Def: Azt a programot, ami a számítógép-programozás megkönnyítésére, illetve részben automatizálására szolgál, **integrált fejlesztői környezetnek** (ang. *Integrated Development Environment*, röviden ***IDE***) nevezzük. □

Fontos megjegyezni, miért is van szükségünk az integrált fejlesztői környezetekre. Már volt említve, hogy a számítógépek csakis és kizáróan *gépi nyelven* értenek, ami viszont az ember számára teljesen idegen. Ezért lettek kitalálva az alacsony (*Assembly*) és magas (*C*, *C++*, *Pascal*, *Java*, *C#*, stb.) szintű programnyelvek. Az emberek ma az esetek túlnyomó többségében magas szintű programnyelven programoznak. Amikor az ember megír egy ilyen programot (valamilyen magas szintű programnyelvben), ezt a programot **forrásprogramnak** vagy **forráskódnak** (ang.

Source Program, *Source Code*) nevezzük. De mivel ez a program a számítógépnek érthetetlen, ezt le kell fordítani az ő nyelvére. Ezért születtek meg az *IDE*-k.

A legtöbb *IDE* tartalmaz legalább egy egyszerű szövegszerkesztőt (*editor*) a program megírására, egy fordítóprogramot és sokszor egy összekötő (összeszerkesztő) programot is (programnyelvtől függően). Azonban sok *IDE* tartalmaz hibakeresőt (*debugger*), illetve egyéb hasznos eszközt is. Érdemes megjegyezni, hogy egy meghatározott *IDE* csak pár magas szintű programnyelvet támogat, néhány esetben pedig csak egyet.

A program **előkészítését** (az írástól a futtatásig terjedő időszakaszt) legalább három fázisra bonthatjuk szét:

1. **A program megírása** – habár erre a célra használhatunk általános rendeltetésű szövegszerkesztőt is (pl. *Notepad*)[3], szinte mindegyik *IDE* alapvető része a **szövegszerkesztő** (ang. ***Editor***), ezért a program írását ajánlatos ebben végezni.[4] A forráskód elkészültével azt le kell menteni a merevlemezre, vagy más külső memóriaegységre. A *C* programnyelven írt forrásprogramok kibővítése, azaz kiterjesztése `.C`. Például a `prog` néven elmentett forrásprogramunk teljes neve `prog.c` lesz.

2. **A program fordítása** – minden *IDE* szerves része a **fordítóprogram**, aminek feladata a forráskód gépi nyelvre való lefordítása. A fordítóprogram attól függően, melyik nyelvről kell fordítania gépi nyelvre, lehet *Assembler*, (igazi) fordítóprogram, vagy értelmező. Az ***Assembler*** az *Assembly* fordítója. Ez a legegyszerűbb faja a fordítóknak, hiszen a gépi és *Assembly* nyelv grammatikája egy és ugyanaz, csak az ábécé, illetve a műveletek kódja más. A magas szintű nyelvek ún. **(igazi) fordítókat** (ang. ***Compiler***, de magyarul is gyakran írják **kompajlernek**) vagy ún. **értelmezőket** (ang. ***Interpreter***, de magyarul is gyakran így hívják) használnak. A *kompajler* és az *interpreter* között az a különbség, hogy míg a *kompajler* egyszerre fordítja az egész programot, addig az *interpreter* soronként halad[5].

   Ami a *C* programnyelvet illeti, ennek a nyelvnek *kompajlere* van. A fordítás (*kompajlírozás*) eredménye az ún. **tárgyprogram**, vagy **tárgykód** (ang. *Object Program*, *Object Code*), aminek kiterjesztése `.O`. Így a `prog.c` nevű programunk kompajlírozása után kapunk mellette egy `prog.o` nevű fájlt is.

   Természetesen csak azok a programok fordíthatók le, amik hibátlanok, azaz nincs bennük egy nyelvtani hiba sem. Ha ez nem így van, a fordítóprogram leáll, és hibaüzenetet ír ki. Sok *IDE* editora segítésképpen tartalmazz egy ún. *on-the-fly* hibaellenőrzőt, ami már a program írása közben aláhúzza a potenciális hibákat.

3. **A program összekötése, vagy összeszerkesztése (linkelése)** – a fordítás által született tárgykód, habár már gépi nyelven van, még mindig nem alkalmas a futtatásra. Ugyanis a program írása közben sokszor felhasználunk (azaz hívunk) már kész programokat, ezzel is megkönnyítve a munkánkat. Ezeket viszont most össze kell kötni a tárgyprogrammal. Ezzel a feladattal foglalkozik az **összekötő** (ang. ***Linker***, de magyarul is legtöbbször így nevezik). A linkelés eredménye a **futtatható program** vagy **futtatható kód** (ang. *Executable Program*, *Executable Code*).[6]

---

3 Érdemes megemlíteni, hogy programokat csak sima szövegszerkesztőben érdemes írni, komoly szövegszerkesztőben (pl. *Microsoft Wordpad*, *Microsoft Word*, *LibreOffice Writer*) nem. A számítógépnek ugyanis semmit sem jelentenek a formázott betűk, betűtípusok, keretek, bekezdések, képek, táblázatok.

4 Természetesen az *IDE*-k editora többre képes, mint egy sima *Notepad*, pl. képes automatikusan különböző színnel jelölni a gépi műveleteket, változókat, konstansokat, stb. Ezzel könnyebben áttekinthető lesz a forráskód.

5 Tehát amikor az *interpreter* lefordít egy sort, azt rögtön futtatja, aztán lefordítja a következő sort, és így tovább.

6 Sok *IDE* a fordítás után automatikusan linkel is, ezért sok helyen már nem is látni külön parancsot a linkelésre, hanem csak a fordításra.

A *C* programnyelv esetében a linker megkeresi az `.O` kiterjesztésű programot és linkeli a többi szükséges tárgykóddal. A linkelés végeredménye egy futtatható program, aminek kiterjesztése `.EXE`. Így a `prog` nevű programunk tárgykódjának összekötése után megkapjuk a `prog.exe` nevű futtatható programot.

A programnyelvek szintaxisa és szemantikája

Minden nyelv alapvető része a nyelvtan és a szókincs. Ezek nélkülözhetetlenek ahhoz, hogy helyesen tudjunk másokkal kommunikálni. Így például a természetes nyelvekben a **nyelvtan** vagy **grammatika** a nyelv elemeivel, szerkezetével, törvényszerűségeivel foglalkozó tudomány. A nyelvtannak több területe van, számunkra kettő fontos: a szintaxis és a szemantika. A **szintaxis** vagy **mondattan** a szavak mondatban betöltött szerepével és a szórenddel foglalkozik, tehát azt taglalja, hogyan kell szavakból szószerkezeteket és mondatokat képezni. Ezzel szemben a **szemantika** vagy **jelentéstan** a szavak jelentésével foglalkozik. Természetesen a nyelv építőkövei a szavak, amik halmaza alkotja a nyelv **szókincsét**.

Szinte ugyanez elmondható a programnyelvekről is. A programnyelveknek is van szókincse, ezek azok a szavak, amiknek értelmük és jelentésük van az adott programnyelvben. Természetesen ezeket tudni kell megfelelően írni, tudni kell egymással összekötni, és tudni kell belőlük értelmes mondatokat szerkeszteni. Azokat a szabályokat, amikkel egy programnyelvben helyes nyelvi konstrukciókat képezhetünk, **szintaxisnak** nevezzük.

Ezzel szemben egy programnyelvben a **szemantika** azt árulja el, mi bizonyos nyelvi konstrukcióknak az értelme, jelentése.

Ahhoz, hogy egy programnyelvet megértsünk, nagyon jól kell ismernünk annak szintaxisát és szemantikáját is. Ugyanis egy programnyelvben (a természetes nyelvekkel ellentétben) nem lehetnek kétértelműségek és félreértelmezések. Pont ezért jöttek létre a szintaxis elsajátítását megkönnyítő formális szintaxisdiagramok.

Szintaxisdiagramok

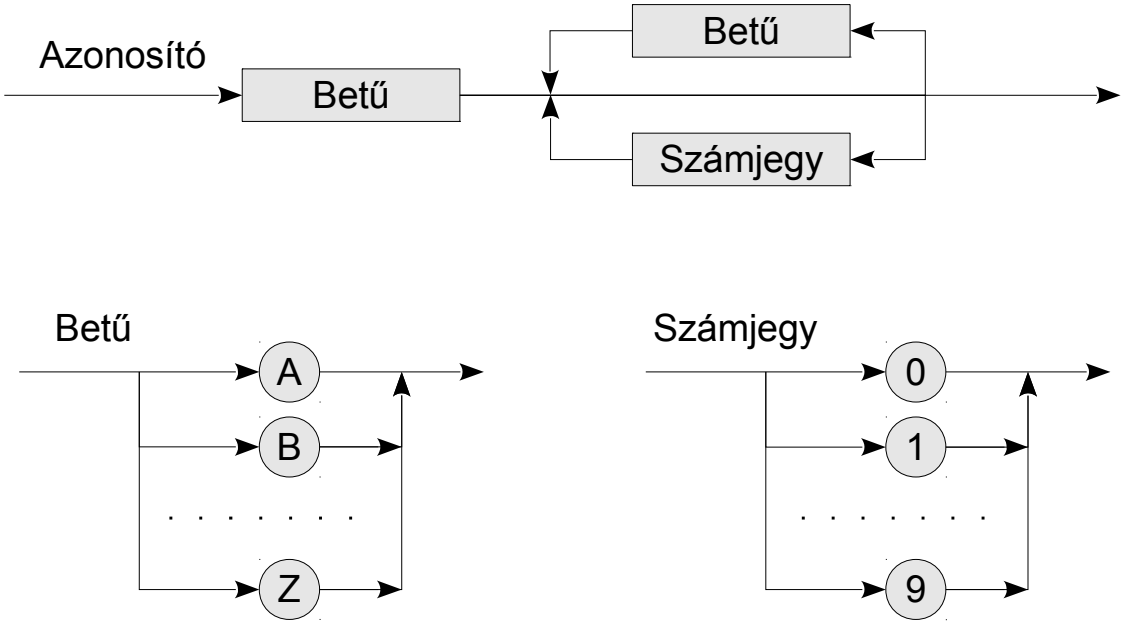
Def: A **szintaxisdiagram** (ang. *Syntax Diagram*) nem más, mint egy adott programnyelv szintaktikailag helyes szószerkezeteinek és mondatainak vizuális módon történő képzése. □

A szintaxisdiagram valójában egy igen egyszerű konstrukció. Lényegében egy blokkokból álló útvonalat mutat. Kétféle **blokk** létezik:

- **Nem terminális szimbólum** – olyan blokk, ami *tovább bontható* valamilyen újabb szintaxisdiagramra. Az *összetettebb* konstrukciókat nem terminális szimbólumként jelöljük. A nem terminális szimbólum grafikai jelölése a **négyzet**, vagy **téglalap**.
- **Terminális szimbólum** – olyan blokk, ami már *tovább nem bontható* újabb szintaxisdiagramra. A programnyelv *alapelemeit* terminális szimbólumként jelöljük. A terminális szimbólum jelölése a **kör** vagy **ellipszis**.

Ezeket a blokkokat **nyilak** kötik össze, amik azt az **útvonalat** jelölik, amin „legálisan” végighaladhatunk úgy, hogy szintaktikailag helyes szerkezeteket képezzünk. Más szóval, ha helyes szintaxisú szószerkezeteket szeretnénk képezni, csak végig kell haladni az útvonalon.

Egy szintaxisdiagramot legjobb egy példával szemléltetni. Tételezzük fel, hogy egy bizonyos programnyelvben az azonosítót leíró módszerrel a következőképp lehetne elmagyarázni: az *azonosító* egy olyan alfanumerikus karakterek (tehát számok és betűk) sorozata, ami mindig betűvel kezdődik. Az azonosító szintaxisdiagramja a képen látható.



1. Kép: Az „azonosító” szintaxisdiagramja

Mint látjuk, ez a szintaxisdiagram tökéletesen helyettesíti az előbb leírt mondatot. Maga az „azonosító” nem terminális szimbólumokból áll, hiszen a „betű” és „számjegy” tovább bontható újabb szintaxisdiagramokra. Ne feledjük: ezekre a diagramokra építhetünk, hiszen létezik olyan programnyelvi szerkezet, ami az „azonosítót” építőelemként használja. Ekkor nem kell újra definiálni az „azonosító” fogalmát, hiszen már egyszer lerajzoltuk.

A szintaxisdiagramok kitűnő alternatívát képeznek a programnyelvek szintaxisának elsajátítására. Vannak emberek, akik könnyebben megjegyzik, ha saját szavaikkal magyarázzák el a dolgokat, nekik megfelel a szintaxis szöveges magyarázata. Viszont mások inkább „vizuális” módon memorizálnak könnyebben, és nekik lehet ideális a szintaxisdiagram. Azonban sok szakirodalomban kombinálják a két módszert, hiszen vannak olyan összetettebb szerkezetek, amiket nehéz elmagyarázni szavakkal (anélkül, hogy kétértelműségek jönnének létre), de van eset, amikor a fordítottja az igaz (amikor a szintaxisdiagram túl komplikált, vagy sok helyet foglal el).

2. fejezet: Algoritmusok

Ennek a fejezetnek az a célja, hogy megértesse az olvasóval, mi az algoritmus, mi az összefüggés algoritmus és számítógépes program között és miért fontos az algoritmusos gondolkodás elsajátítása a programozás világában. Először magát az algoritmus fogalmát fogjuk definiálni, majd rávilágítunk, mi az összefüggés egy probléma megoldása és az algoritmus között. Ezután megadjuk a program egy alternatív definícióját, de most az algoritmus fogalmát felhasználva. A fejezet talán legfontosabb része az algoritmus grafikus ábrázolásának bemutatása lesz, az úgynevezett folyamatábrák. Ez azért fontos, mert a legtöbb ember számára, aki most próbálkozik először programozással, sokkal egyszerűbb először egy problémát vizuálisan folyamatábra segítségével megoldani, mint egy konkrét programozási nyelvben. Ezután – egy kis szünetet tartva – visszatérünk egy tiszta elméleti részhez: először a feladat elemzéséről lesz szó, majd a feladat megoldásának négy fő fázisáról, végül röviden azt fogjuk ismertetni, hogyan lehet egy algoritmus helyességét leellenőrizni.

Az algoritmus definíciója és tulajdonságai

Az életben már rengetegszer kerültünk olyan szituációba, hogy valamilyen problémát kellett megoldanunk, legyen az egyszerű, mint a teafőzés, telefonálás, cipőfűzés, vagy összetett, mint valamilyen komplikált matematikai feladat megoldása. Azonban kevesen veszik észre, hogy mindezek a problémák kisebb részekre oszthatók, amik megoldása már jóval egyszerűbb. Ezek a kisebb problémák sokszor még kisebb részekre oszthatók, egészen addig, míg el nem érünk egy olyan problémához, ami már tovább nem osztható, vagy annak nincs értelme.

Azt is észre lehet venni, hogy valamilyen problémát nem csak kisebb részekre lehet bontani, hanem azt is, hogy ezeket a részeket meghatározott sorrendben lehet csak végrehajtani. Ezért a probléma megoldásának előbb említett kisebb részeit úgy is felfoghatjuk, mint lépések. Habár léteznek olyan problémák, amik megoldásánál annyira nem számít a lépések szigorú sorrendje, minél összetettebb a probléma, annál nagyobb figyelmet kell rá szentelni. Természetesen sorrend alatt nem azt értjük, hogy egy pontból csak egy meghatározott úton tudunk egy másik pontba jutni, inkább azt, hogy minden lépés után pontosan tudjuk, melyik lépés fog (vagy melyik lépések fognak) következni, vagyis egy pontból akár többféle úton (módon) is haladhatunk a másik pontig, míg az precízen definiálva van. Amikor egy összetett problémát részekre, lépésekre bontunk, mi lényegében annak algoritmusát dolgozzuk ki.

Def: az **algoritmus** olyan megengedett lépésekből álló módszert, eljárást, receptet jelent, amely alkalmas valamely felmerült probléma vagy feladat megoldására. □

Vegyünk egy egyszerű problémát, például a telefonálást, és próbáljuk meg azt lépésekre bontani. A lépések a következők:

1. Felvenni a telefonkagylót és megvárni a jelzést, ami jelzi, hogy szabad a tárcsázás.
2. Beírni a hívni kívánt fél telefonszámát a telefonba és megvárni, míg a hívott fél fel nem veszi a telefonkagylót.
3. Elvégezni a beszélgetést.
4. Letenni a telefonkagylót.

Természetesen ez egy nagyon primitív példa, és a megoldás is hagy maga után kívánnivalót. Például észre lehet venni, hogy az első és második lépést tovább bírnánk bontani kisebb egységekre. Továbbá, ha jobban belemerülünk a problémába, akár azt is észrevehetjük, hogy az egész folyamat felborulhat, például a második lépésnél, amennyiben a tárcsázott szám foglalt, vagy senki sem jelentkezik. Tehát nem hogy a lépéseket kisebb részekre bonthatjuk, hanem magát a megoldást is bővíthetjük új lépésekkel, kitérve azokra az esetekre, amikre nem mindig számíthatunk, de ha reálisan nézzük, bekövetkezhetnek.

Amikor az algoritmus lépéseiről beszélünk, fontos megjegyezni, hogy minden lépésnek egyszerűnek kell lennie. Ez azért fontos, hogy ne lépjenek fel kétértelműségek, de az se történjen meg, hogy a lépést ne tudjuk végrehajtani, mert túl komplikált. Ha úgy érezzük, hogy fennáll az előbb említett veszélyek egyike, akkor kénytelenek leszünk a lépést kisebb részekre osztani. Persze továbbra is fel lehet tenni a kérdést, miért kell egy lépést kisebb részekre osztani, ha az számunkra teljesen érthető. Azért, mert az algoritmus lényegében egy olyan lépéssorozat, amit mindenkinek végre kell tudni hajtani, még egy olyan személynek is, aki azt sem tudja, mi maga a probléma. Tehát, bárkinek is adjuk oda az algoritmust, hogy menjen rajta keresztül, végig tud majd rajta haladni, és bárkinek is adjuk oda az algoritmust, mindenki ugyanazt a megoldást fogja kapni.

Ha már az algoritmus egy olyan lépéssorozat, amit mindenki végre tud hajtani, még akkor is, ha a személy azt sem tudja, mi maga a probléma, akkor a lépéssorozat végrehajtását odaadhatjuk akár egy gépnek is. A számítógép egy ilyen gép.

Azonban ahhoz, hogy számítógépet használjunk valamilyen probléma megoldására, nem elég azt betanítani az alapvető műveletek helyes elvégzésére. Fontos az is, hogy tudjuk, a problémának létezik megoldása, és a megoldást véges idő alatt lehet elérni. Ha ez nem így van, lehet, hogy a számítógép soha nem fog az algoritmus végére érni.

Amikor a számítógép egy algoritmust old, sokszor szüksége van olyan információkra, vagy adatokra, amiket a számítógép nem tud saját maga kitalálni. Például egy matematikai példa megoldásánál tudnia kell, melyik számokkal kell dolgoznia, mielőtt elkezdi a megoldás számolását. Ezért az algoritmusoknak kell, hogy legyen:

- **Bemenete** – azok az adatok, információk, amiket meg kell adni a számítógépnek az algoritmus végrehajtásának elején vagy folyamán. Az adatok száma magától az algoritmustól függ (létezik olyan eset is, amikor nincs szükség bemeneti adatokra).
- **Kimenete** – az algoritmus végrehajtásának eredménye (vagy eredményei), ami összhangban áll a megadott bemeneti adatokkal.

Egy algoritmusban megadhatjuk, melyik bemeneteli adatok lehetnek „legálisak”, azaz engedélyezettek (például a számok engedélyezettek, a betűk viszont nem). Azonban ne feledjük, az algoritmus egy *általános* lépéssorozat, tehát minden *engedélyezett* bemeneti adatra helyes eredményt kell generálnia. Ha valamilyen bemeneti adatra mégsem kapunk jó eredményt (kimenetet), akkor vagy nem korlátoztuk le eléggé az engedélyezett bemeneti adatok számát, vagy rossz az algoritmus. Az algoritmus általánossága fontos, hiszen akkor szoktuk a számítógépet valamilyen probléma megoldására hívni, amikor az az embernek túl bonyolult. Amikor egy algoritmus minden engedélyezett bemeneti adatra helyes kimenetet ad, akkor azt mondjuk, hogy az algoritmus **alkalmazható** (ellenkező esetben pedig azt, hogy **nem alkalmazható**).

A feladat és az algoritmus

Természetesen amikor az ember egy problémát, egy feladatot akar megoldani, legtöbbször nem vesz tudomást arról, hogy a belőle készíthető algoritmus egy precíz lépéssorozatot rejt, sem arról, hogy az algoritmusnak van bemenete és kimenete. Azonban ezek fontosak ahhoz, hogy az algoritmus végrehajtását a számítógépre bízhassuk. Tehát, amikor egy feladatot meg szeretnénk oldani, annak folyamata a következőképp kell, hogy kinézzen: ha adott maga a feladat (probléma) és az engedélyezett bemeneti adatok halmaza, akkor készítsük el azt az algoritmust, ami megoldja ezt a feladatot.

Amikor az ember létrehoz egy algoritmust, csak kevés esetben éri be azzal, hogy az algoritmus *alkalmazható*. Legtöbbször felteszi magában a kérdést, lehetne-e ugyanezt a feladatot egyszerűbben, elegánsabban, „olcsóbban", hatékonyabban, gyorsabban is megoldani. Egy algoritmus optimalitásával, hatékonyságával az algoritmusok **minőségi mutatója** foglalkozik. Itt több megközelítés létezik:

- Van olyan megközelítés, ami azt nézi, hány *lépésből* áll az algoritmus – minél kevesebből, annál jobb.
- Egy másik azt nézi, mennyire *gyorsan* lehet az egyes lépéseket végrehajtani – minél gyorsabban, annál jobb.
- Egy harmadik azt nézi, mennyi részeredmény fog születni az algoritmus végrehajtása közben (*memóriahasználat* mértéke – minél kevesebb, annál jobb).

Hogy melyik megközelítést részesítjük előnyben, az főleg attól függ, mi számunkra a fontos, tehát hogy milyen szempontból szeretnénk algoritmusunkat jobbá tenni.

Azonban ahhoz, hogy átadhassuk a probléma megoldási módját egy számítógépnek, el is kell azt neki magyarázni. Ahhoz, hogy egy ember megtanítson egy másik embert valamilyen algoritmusra, elég azt leíró módon elmagyarázni, hasonlóképp, mint az előbb említett telefonálós példánál. A számítógépeknél ez nem fog működni, mivel, mint tudjuk, számukra az emberi nyelv érthetetlen. A fordítottja is igaz, a legtöbb embernek meg a gépi nyelv érthetetlen. Ezért lettek kitalálva a programnyelvek. A programnyelvek úgy lettek felépítve, hogy relatív egyszerűen lehessen segítségükkel az algoritmus lépéseit a számítógépnek elmagyarázni. Nem nehéz rájönni, hogy a számítógép gépi utasításai valójában az algoritmus alapvető lépései, míg az egész algoritmus lényegében a program. Tehát a programot a következőképp is definiálhatjuk:

Def: azt a számítógép számára is érthető algoritmust, amit valamilyen programnyelvben írtunk meg, **programnak** nevezzük. Tehát, egy számítógépnek a program nem más, mint az algoritmus ábrázolásának egy módja. □

A feladat elemzése

Az összetett problémáknál, feladatoknál, még mielőtt belekezdenénk az algoritmus létrehozásához, fontos magának a feladatnak az elemzése. A **feladat elemzése** lényegében a feladattal való megismerkedés. Tehát, itt el kell gondolkozni azon, mi is a probléma, mi annak a lényege, mit kapunk azáltal, hogy megoldjuk a problémát, és mi az, amit ismerünk a feladat elején. Ez alapjában véve megegyezik azzal a feltevéssel, hogy mindegyik feladat három komponensből épül fel: bemeneti adatokból, adatfeldolgozásból (transzformációkból) és kimeneti adatokból

(eredményekből). Természetesen a három komponens közül a második megtalálása a legnehezebb, ugyanis itt kell kitalálni azt az algoritmust, ami majd megoldja a problémát.

Az olyan egyszerű feladatoknál, mint amilyenekkel mi fogunk foglalkozni, nem nehéz kitalálni az algoritmust, és ezért a probléma elemzése sem tekinthető valami bonyolultnak. Viszont a valóságban a komoly programok írása előtt a cégekben egész csapatok foglalkoznak a feladat elemzésével, akár több hónapig is. Ebben az időszakban felállítják a problémát, a célokat, felvázolják, milyen részekből fog a megoldás állni, definiálják a részek közti kapcsolatokat, egyszóval létrehozzák a megoldás konstrukcióját. Közben mindent részletesen dokumentálnak is. A számítástechnikai egyetemeken egész szakok foglalkoznak ezzel a problematikával.

A feladatok megoldásának fázisai

Egy feladat megoldásának folyamata lehet relatív egyszerű, és lehet nagyon összetett is, attól függően, hogy milyen nehézségű feladatot kell megoldani. Már említettük az előző fejezetben, hogy a programozás egy lényegében három fázisból álló tevékenység. Ide soroltuk a tervezés fázisát, a program írását és a tesztelést. Nos, a feladatok, problémák megoldásának fázisai alapjában véve megegyeznek az előbb említett fázisokkal, annyi különbséggel, hogy most a tervezés fázisa két részre lesz bontva, és mindegyik fázis bővebben be lesz mutatva. Tehát, a **feladatok megoldásának fázisai** a következők:

1. **A feladat megfogalmazása** – ahhoz, hogy bármilyen feladatot meg tudjunk oldani, először meg kell azt érteni. Dokumentálni kell, hogy mit ismerünk a feladatból, azaz mit tudunk felhasználni bemeneti adatként, és mit kellene kapni megoldásként. Ez a fázis elég rövid és egyszerű lehet a triviálisabb feladatoknál, viszont az összetett problémáknál sokáig elhúzódhat. Az összetett feladatoknál ajánlatos a feladat precíz *elemzése*.

2. **Az algoritmus megtervezése, létrehozása** – a feladat megfogalmazása, elemzése után neki lehet kezdeni az algoritmus tervezéséhez. Ez egy igen nehéz fázis, hiszen itt kell lényegében kitalálni, hogyan lesz a probléma megoldva. Az egyszerű problémák algoritmusát sokszor elég csak fejben tartani, de a már kicsit komplikáltabb feladatoknál szükséges annak dokumentálása, lejegyzése. Itt több opció áll rendelkezésünkre: használhatjuk a *szabálykészlettel* való ábrázolást, a *folyamatábrákat*, vagy valami egészen mást.[7]

   Érdemes megemlíteni, hogy az algoritmus megtervezése után nem árt, ha azt leteszteljük, hiszen egyáltalán nem biztos, hogy a kitalált algoritmus pontos. Habár ez a folyamat rendelkezésünkre áll később is, több időt meg lehet spórolni, ha már most észrevesszük a hibákat, vagy legalább azok egy részét. Ne feledjük, ha hibás az algoritmus, a program is az lesz.

3. **Az algoritmus megvalósítása, azaz a program írása** – amikor úgy érezzük, hogy az előző fázisban létrehozott algoritmus jól működik, ideje nekilátni annak megvalósításához, ami nem más, mint a **program írása**. Az általunk kiválasztott programnyelv segítségével el kell magyarázni a számítógépnek, hogyan kell a problémát megoldani, azaz át kell fordítanunk az előző fázisban kapott algoritmust a kiválasztott programnyelvre.

4. **A program tesztelése** – ennek a fázisnak az a célja, hogy bizonyosságot szerezzünk azzal kapcsolatban, hogy az algoritmus (és vele együtt a program) pontosan működik, vagy nem. A felmérések azt mutatják, hogy szinte lehetetlen olyan algoritmust és programot kitalálni, ami rögtön a legelején tökéletesen fog működni. Sokszor az is kiderül, hogy a program csak

7 A szabálykészlettel való ábrázolást, valamint a folyamatábrákat a következő fejezetben (*Algoritmusok – gyakorlati rész*) fogjuk ismertetni.

néhány bemeneti adat esetében működik helyesen, a többi esetben nem, habár engedélyezett bemeneti adatokról van szó.

A program megírása után elsőként a **szintaktikus hibákat** kell orvosolni. Ezek azok a hibák, amiket ha nem javítunk ki, el sem indíthatjuk a programot. Szerencsére itt sokat segít a programnyelv fordítója, ugyanis az a legtöbbször jelezni fogja, hogy bizonyos utasításoknál szintaktikai hibát talált, és addig nem engedi a program futtatását, míg ki nem lesz javítva.

A szintaktikai hibák kijavítása csak azt fogja eredményezni, hogy a program futtathatóvá válik, de – mint ahogy sokszor rájön az ember – ez még sajnos nem azt jelenti, hogy a program helyesen is működik. Itt a programnyelv fordítója nem tud segítséget nyújtani, mivel nem tud helyettünk gondolkodni. Ezeket **logikai hibáknak** nevezzük. Ezek a hibák a programnyelv szemantikájának rossz értelmezése miatt is történhetnek, de talán nagyobb a valószínűsége, hogy simán rossz a kitalált algoritmus. A logikai hibákat úgy lehet felismerni, hogy habár a program elindul, az a program futása közben hirtelen megszakad, leragad, vagy rossz eredményt dob ki a bevitt adatokra. A logikai hibák forrásának megtalálása és kijavítása nehéz tud lenni, és gyakran kényszerül az ember arra, hogy magát az algoritmust változtassa.

A tesztelés általában annyiból áll, hogy bizonyos számú **tesztadatot** találunk ki. A tesztadat egy olyan bemeneti adat (vagy azok halmaza), amikre pontosan tudjuk, milyen kimeneti eredményt kell kapni. A program futásakor megadjuk neki az összes tesztadatot, és ha mindegyikre helyes kimenetet ad, akkor azt lehet mondani, hogy a program helyesen működik – a tesztadatokra, de ezzel még nem lehetünk biztosak, hogy a program minden engedélyezett bemeneti adatra jól fog működni. Mivel az engedélyezett bemeneti adatok halmaza óriási lehet, a legtöbb esetben lehetetlen annak egyenkénti letesztelése, ez miatt fontos, hogy a tesztadatok átfogók és reprezentatívak legyenek, azaz tartalmazzanak tipikus bemeneti adatokat is, de *„kritikus”* adatokat is, azaz olyanokat, amiknél van egy olyan érzésünk, hogy baj léphet fel.

Amikor rossz eredményt kaptunk (vagy semmilyent) valamilyen bemeneti adatra, lokalizálni kell a hiba forrását, és ki kell azt javítani. Erre több módszer is létezik, de javarészt a következő ötleten alapszanak. Mint tudjuk, az algoritmus több lépésből áll. Mivel a számítógépek nagyon gyorsak, ezért sokszor egy pillanat alatt megkapjuk az eredményt, miközben lehet, hogy több száz vagy ezer lépés is végre lett hajtva a háttérben. Nekünk ekkor általában az a feladatunk, hogy rájöjjünk, melyik lépés vagy lépések a hibásak. Azonban a programok sokszor úgy vannak kitalálva, hogy a bemeneti adatok megadása után csak kivágják a végeredményt. A hibák lokalizálásánál viszont sokat segíthet, ha tudnánk a bizonyos lépések után kapott részeredményeket is. Ezt az ötletet felhasználva könnyen meg lehet találni, melyik lépésnél történt a hiba. Az *IDE*-k sok esetben tartalmaznak beépített **hibakeresőt** (ang. ***Debugger***). Ez annyiból áll, hogy megjelöljük a program kritikus, illetve hibagyanús lépéseit (utasításait), és a *debugger* a program futása közben a megjelölt lépésnél leállítja egy pillanatra (szünetelteti) a program futtatását, hogy mi megnézhessük az ott kapott részeredményt. Ha a részeredmények helyesek, akkor folytathatjuk a program futtatását a következő megjelölt lépésig, és így tovább. A másik módszer az, hogy *debugger* helyett mi fogjuk kiíratni a képernyőre a részeredményeket. Ez annyiból áll, hogy az algoritmusba (tehát a programba) a kritikus lépések mögé adatkiíró utasításokat szúrunk be, amikkel a program futása közben nem csak a végeredményt, hanem a részeredményeket is kiírathatjuk a képernyőre. Ezt **megjelölő technikának** (ang. *Flagging*) nevezzük.

Fontos, hogy az összes fázist megfelelően **dokumentáljuk**. Ez annyiból áll, hogy a fázisok fontos mozzanatait lejegyezzük. Például az első fázisnál lejegyezhetjük, mi a probléma lényege, mikre kell odafigyelni, aztán az algoritmus készítésénél és a program írásánál megjegyzést írhatunk, hogy miért lettek a leírt módon megszerkesztve egyes lépések[8], a tesztelésnél pedig lejegyezhetjük, milyen tesztadatokat adtunk meg a programnak. Még a rövid feladatokat is érdemes dokumentálni, mivel lehetnek benne első ránézésre érthetetlen lépések. Ne feledjük, a dokumentálás nem csak másoknak, hanem önmagunknak is szól, mivel pár hónap szünet után a programot már tulajdon szerzője sem fogja már érteni.

Az algoritmus helyességének ellenőrzése

Az algoritmus helyességének ellenőrzése talán az algoritmus készítésének legnehezebb és leghosszabb fázisa. Mint ahogy már említettük, ezt az ellenőrzést legtöbbször úgy végezzük, hogy bizonyos számú **tesztadatot** találunk ki, és megnézzük, hogy helyes eredményeket (kimeneti adatokat) kapunk, vagy nem. Természetesen olyan tesztadatokat kell kitalálni, amiknek előre ismertek az eredményei, vagy ki lehet őket gyalog számolni. Sajnos, példák segítségével mi csak inkább azt tudjuk bizonyítani, hogy az algoritmus nem helyes (amikor rossz eredményt kapunk valamely tesztadatra), viszont annak bizonyítása, hogy az algoritmus helyes, már sokkal nehezebb, vagy akár lehetetlen, hiszen még ha sikeresen átmegy az algoritmus az összes tesztadaton, akkor is csak legfeljebb azt mondhatjuk, hogy az algoritmus helyes a tesztadatokra, de azt már nem, hogy az algoritmus globálisan is helyes.[9]

Már volt arról szó, hogy a tesztadatok kitalálásánál fontos, hogy azok reprezentatívak legyenek, tehát tartalmazzanak *tipikus* bemeneteli adatokat is (amiket nagy valószínűséggel a felhasználó is használni fog), de *kritikus* adatokat is (amiket nem biztos, hogy a felhasználó használni fog, de fennáll a gyanú, hogy nem fognak jól működni). Amiről eddig még nem volt szó, az az, hogy a tesztadatoknak **átfogónak** kell lenniük. Ez annyit jelent, hogy ha az algoritmus elágazó, vagy ciklusos (egyszóval, nem lineáris) struktúrával rendelkezik, akkor fontos, hogy a tesztadatokkal végigsétáljunk az algoritmus összes lehetséges „útvonalán”.[10]

8 A programnyelvek igen elegáns megoldást használnak megjegyzések írására. A *C* programnyelv megjegyzéseivel (kommentárjaival) később még foglalkozni fogunk.

9 Már előbb volt róla szó, hogy amikor az algoritmus minden engedélyezett bemeneti adatra helyes kimeneti adatot (eredményt) ad, akkor azt mondjuk, hogy az algoritmus *alkalmazható*.

10 Az elágazó és ciklusos (tehát nem lineáris) struktúrákat a következő fejezetben (*Algoritmusok – gyakorlati rész*) fogjuk ismertetni.

3. fejezet: Algoritmusok – gyakorlati rész

Az előző fejezetben elméleti szemszögből próbáltuk az algoritmusokat ismertetni, most viszont eljött az idő, hogy gyakorlati szemszögből is megismerkedjünk velük. Ennek a fejezetnek az a célja, hogy az olvasó elsajátítsa az algoritmusok vizuális ábrázolásának legismertebb formáját, a folyamatábrákat, majd pedig ezt felhasználva ismerkedjen meg az algoritmusos gondolkodás alapjaival. A fejezet először az algoritmusok vizuális ábrázolását – a folyamatábrákat – fogja bemutatni, majd rátér az algoritmusok három fő struktúrájának magyarázására: ezek a szekvenciális, elágazós és a ciklusos struktúrák.

Az algoritmus grafikus ábrázolása

Minden komplex probléma megoldásának algoritmusát olyan egymást követő lépések sorozatára kell redukálni, amik a keresett eredményhez vezetnek. E célból különböző az algoritmusok ábrázolására szolgáló rendszerek jöttek létre. Minden rendszer két fontos alkotóelemből áll: a **lépésekből (utasításokból)** és azok helyes **sorrendjéből**. Az algoritmus ábrázolásának két általános célja van:

1. A szemléltetett algoritmusnak érthetőnek kell lennie különböző felhasználóknak. Ez lehetővé teszi a már meglévő algoritmusok újbóli felhasználását.
2. A szemléltetett algoritmus alapján megírható valamilyen programnyelvben az algoritmus számítógépes programja. Tehát, annak érdekében, hogy az algoritmust egy számítógép hajtsa végre, szükség van az algoritmus számítógépbe való átvitelére.

Az algoritmusok ábrázolására szolgáló rendszerek közel állnak a számítógépek világához, mégis az emberek használják. Ezért úgy is tekinthetünk rájuk, mint valamiféle „univerzális” nyelvre, amik nem valamiféle programnyelvhez vagy számítógéphez vannak igazítva, hanem az emberhez. Akkor lehetnek hasznosak, amikor az ember még nem ismeri a programnyelveket, vagy amikor univerzálisan, programnyelvtől függetlenül szeretnénk valamilyen algoritmust ábrázolni.

Több ilyen rendszer létezik. Egyik ilyen rendszer az algoritmusok szabályok sorozatával (azaz **szabálykészlettel**) való ábrázolása. Ekkor az algoritmust *leíró* módon, saját szavainkkal szemléltetjük, hasonlóképp, mint egy receptet. A lépéseket szabályok formájában írjuk le, amik *sorszámozva* vannak, és minden sorba egy szabályt (lépést) írunk. A szabályok sorszáma mutatja, milyen sorrendben kell a lépéseket végrehajtani. Mivel fennáll a kétértelműség veszélye, ezért az összetettebb szabályokat nem teljesen emberi nyelven írjuk, hanem inkább oly módon, ami inkább valamilyen programnyelv nyelvtanához áll közel. Így kicsit furcsán hatnak a mondatok, de sokkal „biztonságosabbak”.

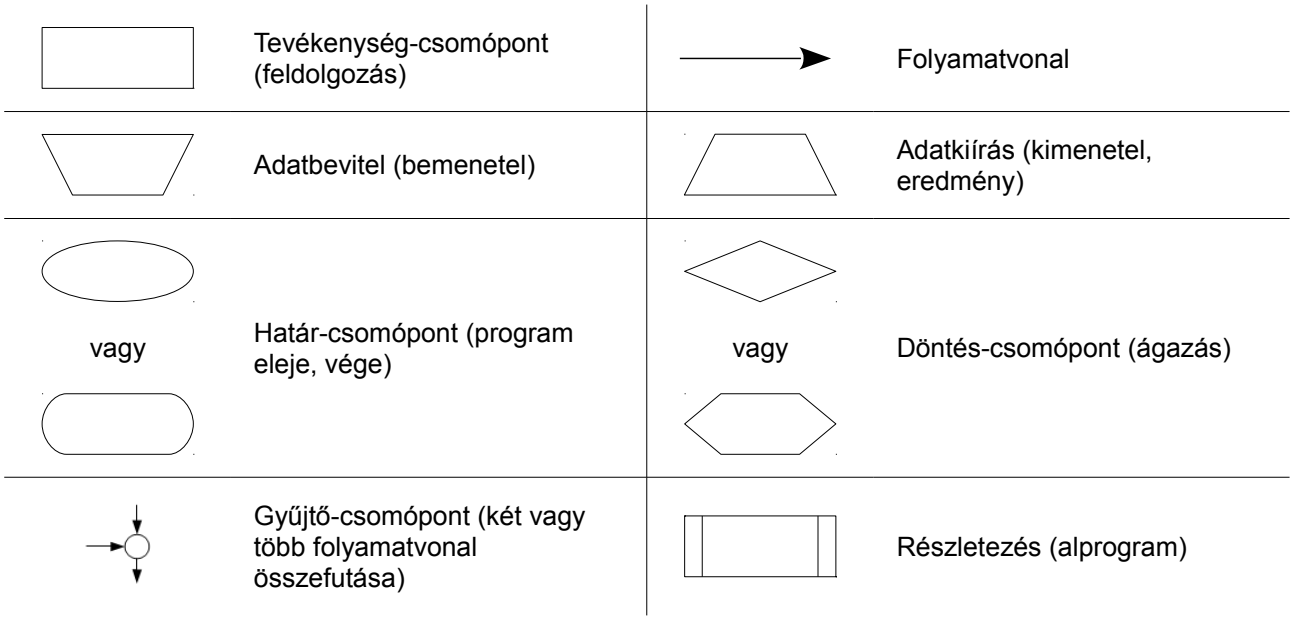
A fejezet elején bemutatott telefonálás példája eléggé hasonlít erre a rendszerre. Példaként vegyünk egy másik feladatot, pl. írjunk egy algoritmust, ami kiszámolja a megadott x szám ötszörösét, majd kiírja (kinyomtatja) az x számot, valamint a kapott eredményt. Az algoritmus szabálykészlettel való ábrázolása a következőképp nézne ki:

1. Az x szám beírása.
2. Kiszámolni az x szám ötszörösét ($5x$).
3. Kiírni az x és az eredmény ($5x$) értékét.

Mi a szabálykészlettel való ábrázolással nem fogunk bővebben foglalkozni. Helyette egy másik rendszerre fogunk összpontosítani, amivel vizuális, grafikus módon tudjuk az algoritmusokat ábrázolni. Ezt a rendszert **folyamatábrának** (ang. *Flow Chart*) nevezzük. Mivel egy vizuális ábrázolásmódról van szó, ezért itt a rendszer két alkotóelemét – a lépéseket és azok sorrendjét – is grafikus módon kell szemléltetni, méghozzá valamilyen grafikus szimbólumokkal. Ebben az esetben az algoritmus lépéseit (utasításait) különböző geometriai alakzatokkal, ún. **blokkokkal** ábrázoljuk (pl. téglalap, trapéz, ellipszis stb.), míg a a lépések sorrendjét a blokkok megfelelő módon való összekötésével, tehát **nyilakkal**. Azért létezik többféle blokk, mert az utasításoknak is több típusa van: például vannak adatbeíró, adatkiíró, adatfeldolgozó blokkok, stb. A blokkokat sokszor **csomópontoknak** is nevezzük, míg az azokat összekötő nyilakat **folyamatvonalnak**.

Ma már a folyamatábráknak is több formája, fajtája ismert, mi ebből egy régebbi, relatív egyszerű fajtát fogunk ismertetni.

A folyamatábra általános elemei a következők:



A legalapvetőbb szimbólumok a következők:

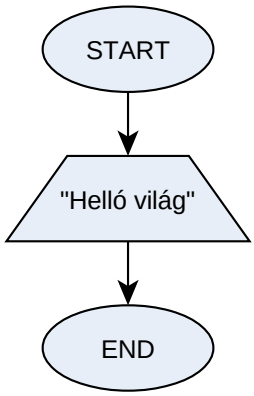
- A *téglalap* formájú **tevékenység-csomópontban** végezzük az *adatfeldolgozást*, pl. a számolást.
- A *tölcsér alakú trapézban* végezzük az **adatok beírását**. Legtöbbször akkor használjuk, amikor a felhasználót arra szeretnénk kérni, hogy adja meg (írja be) egy változó értékét, legtöbbször a billentyűzeten keresztül. A tölcsér formájú trapéz szimbólumot könnyű megjegyezni, hiszen úgy néz ki, mint egy tölcsér, amibe bedobhatjuk a feldolgozandó adatokat.
- A fenti csomópont ellentéte a *fordított tölcsér alakú trapéz*, az **adatkiírás**, ami a feldolgozás eredményének *kiírására* szolgál, legtöbbször a képernyőre.
- Az algoritmus *elejét* és *végét* az *ellipszis* vagy *lekerekített téglalap* formájú **határ-csomóponttal** jelöljük. Ezek jelképezik az algoritmus elejét és végét.
- A fenti csomópontokat természetesen *nyíl* formájú **folyamatvonalakkal** kötjük össze, ezzel egy útvonal alakul ki, amin végig lehet haladni. Ahhoz, hogy az algoritmust

végrehajthassuk, az algoritmus elejét jelző határ-csomóponttól indulunk, követjük az utat a nyilak segítségével, közben végrehajtva a csomópontokat. Az algoritmus végrehajtása az algoritmus végét jelző határ-csomópontnál fejeződik be.

Jó ideig csak ezzel a pár blokkal fogunk dolgozni, aztán viszont hozzá fogjuk venni a következőket is:

- A *rombusz*, vagy *hatszög* formájú **döntés-csomópontot** *elágazások feltételeinek* jelölésére használjuk, tehát amikor a folyamat több útra szakad. Ugyanis hamarosan rá fogunk jönni, hogy nem csak egy úton juthatunk célba, hanem akár több úton is.
- Minden döntés-csomópontnál, amikor az út kettészakad, szükség van arra is, hogy ezek az utak egyszer ismét *összefolyjanak*. Ekkor lesz szükség a **gyűjtő-csomópontra**. Jelölése egyszerű: *kis kör*, amibe összegyűjtjük az összefolyó folyamatvonalakat.
- Végül eljutunk a **részletezés** csomópontjába is, amivel az *alprogramokat*, *függvényeket* jelöljük. Ennek magyarázatára majd később fogunk kitérni.

Példaként vegyük a következő egyszerű problémát. Készítsünk egy algoritmust, ami kinyomtatja, kiírja a *„Helló világ”* üzenetet (pl. a képernyőre).



Mint látjuk, minden algoritmus elejét és végét a *START* és *END* határ-csomópontokkal határoljuk be, ezzel jelöljük a program elejét és végét. Természetesen az *END* helyett használhatjuk a magyar *VÉGE*, vagy akár a *STOP* kifejezést is. Mivel az algoritmus csak egy üzenetet ír ki a képernyőre, ezért a fordított tölcsér alakú adatkiírás csomópontját kell használni.

A folyamatábrák elsajátítása kulcsfontosságú, hiszen a hamarosan bemutatásra kerülő algoritmusstruktúrák pont folyamatábrák segítségével lesznek bemutatva.

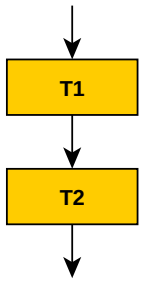
Az algoritmusok struktúrája (szerkezete)

A folyamatábrák ismertetésénél szó volt arról, hogy az algoritmus szerteágazó struktúrával is rendelkezhet. Ez azért van, mert az egyszerű, fentről lefelé haladó, elágazás nélküli programokkal nem sokra lehet menni. Be lehet bizonyítani, hogy egy procedurális programnyelvben szinte minden feladatot, problémát megoldhatunk háromféle programstruktúra használatával. Ezeket **általános algoritmusos struktúráknak** nevezzük, és lehetnek:

- **Szekvenciális (lineáris) struktúrák,**
- **Elágazó struktúrák és**
- **Ciklusos struktúrák.**

Szekvenciális (lineáris) struktúra

A **szekvenciális** vagy **lineáris** struktúra (2. Kép) a legegyszerűbb a három struktúra közül. Amint azt a képen látni lehet, véges számú **lépésből** (**tevékenységből**, röviden *„T”*) épül fel, tehát *T1*, *T2*, …, *Tn*, ahol *n* a lépések száma. A szekvenciális struktúrán, ahogy neve is mondja, szekvenciális, azaz lineáris módon kell végighaladni: először végrehajtjuk a *T1* tevékenységet (lépést), majd azután rátérünk a *T2* végrehajtására, és így tovább, egészen *Tn*-ig. Természetesen minden lépést csak **egyszer** hajthatunk végre.



2. Kép: Szekvenciális (lineáris) struktúra

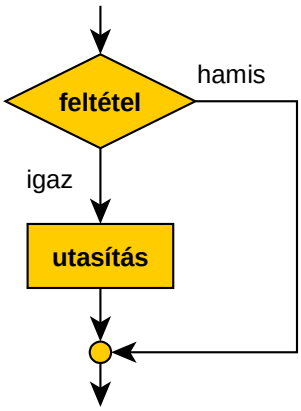
A csak szigorúan lineáris struktúrával azonban kevés hasznos programot lehet írni, ezért a már kicsit bonyolultabb programoknál szükségünk lesz a másik kettő struktúrára is.

Elágazó struktúra

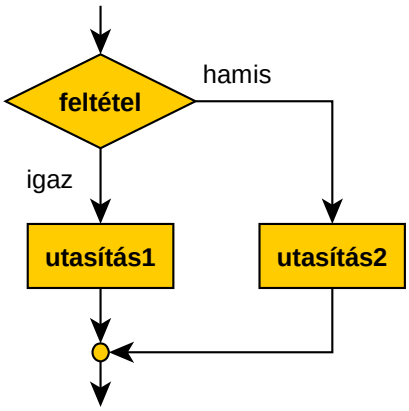
Nagyon sok algoritmus úgy van felépítve, hogy a lépések közül lesznek olyanok, amiket egyetlenegyszer sem fogunk végrehajtani. Ezeket **elágazással** érhetjük el a legkönnyebben. Mint amikor a valós életben egy úton haladva elágazás elé érünk, döntenünk kell: vagy az egyik úton fogunk végighaladni, vagy a másikon. Maga a döntés úgy történik, hogy felmérjük, melyik út felel meg nekünk jobban, melyik út a jobb. Természeten magától értetődik, hogy a nem kiválasztott útvonal ismeretlen marad számunkra, ugyanis visszafordulni már nem tudunk. Ugyanígy van az algoritmusoknál is. Annyi a különbség, hogy a döntésnél mindig valamilyen számolás történik: valamilyen értékeket hasonlítunk össze logikai módon (pl. nagyobb, kisebb, egyenlő, különböző, stb.). Ezt nevezzük **feltételnek** (*„F”*), aminek két értéke lehet: **„igaz”**, vagy **„hamis”**. Attól függően, hogy milyen bemeneti adatokat adunk meg a programnak, a számolás értéke egyik lesz a két válasz közül: ha a válasz „igaz”, akkor az „igaz úton” haladunk tovább, ha pedig a válasz „hamis”, akkor a „hamis úton” folytatjuk utunkat.

Az algoritmusoknál a feltételt **döntés-csomópontba** tesszük, amiből aztán két út (az „igaz” és a „hamis”) indul különböző irányokba. A döntés-csomópont lehet egyágú és kétágú. Az **egyágú** döntés-csomópontnál (3. Kép) a „hamis” út *„üres”*, tehát nem tartalmaz egyéb lépést, hanem csak átugorja azokat a lépéseket, amik az „igaz” ágon találhatók (a képen csak egy lépés van az „igaz” ágon, a *T1*). Ezzel szemben a **kétágú** döntés-csomópont (4. Kép) valódi két ágat tartalmaz, és mindkét ágon találhatók konkrét lépések (a képen *T1* az „igaz” ágon, *T2* a „hamison”).

Természetesen mindkét típusú döntés-csomópontnál az ágak tetszőleges számú lépést tartalmazhatnak, nem csak egyet, mint ahogy a képeken látható.



3. Kép: Egyágú döntés-csomópontot használó elágazás



4. Kép: Kétágú döntés-csomópontot használó elágazás

Az elágazó struktúra tehát két ágra bontja az utat, azonban szükség van arra is, hogy ezek az ágak egyszer ismét találkozzanak, azaz összefussanak, összefolyjanak. Ezt a már ismert **gyűjtő-csomóponttal** érhetjük el. Ne feledjük, az algoritmusoknál és a programoknál kötelező az egyszer elvált utak újbóli összefolyása. Más szóval, tilos olyan algoritmust, vagy programot írni, amiben az utak nem fognak előbb-utóbb (legkésőbb a program végéig) összefolyni.

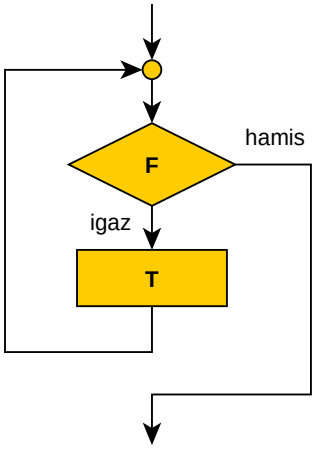
Ciklusos struktúra

Míg az elágazó struktúrák lehetővé tették, hogy bizonyos lépéseket soha ne hajtsunk végre, addig a **ciklusos struktúrák** lehetővé teszik, hogy bizonyos lépéseket többször is végrehajtsunk (vagy egyszer se). Ehhez ki kell választani azokat a lépéseket (tevékenységeket), amiket többször szeretnénk végrehajtani, majd ciklusba zárjuk őket. Ezt nevezzük a **ciklus belsejének**, **testének**, vagy egyszerűen **ciklustestnek**.

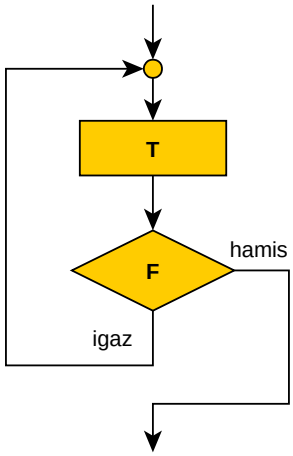
A ciklus végrehajtását *szigorú irányítás* alá kell tenni, különben **végtelen ciklus** alakulhat ki, vagyis egy olyan ciklus, amiből soha nem léphetünk ki. Mivel ez olyan programokat eredményezne, amik az idő végezetéig futnának (kivéve, ha erőszakosan megszakítjuk őket), ezért a ciklustest elé, vagy mögé **feltételt** (*F*) kell tenni. A feltétel itt is ugyanazt a szerepet játssza, mint az elágazásoknál: logikai úton eldöntsük, hogy akarunk-e még egy kört futni a ciklus belsejében, vagy sem. Ezt is **döntés-csomópontba** helyezzük, amiből két ág fog kinőni: az *„igaz”* ág bevezet minket a ciklustestbe, míg a *„hamis”* ág kidob minket a ciklusos struktúrából.

Attól függően, hogy hová tesszük a feltételt (azaz a döntés-csomópontot), a ciklusos struktúra lehet elöltesztelős és hátultesztelős. Az **elöltesztelős** ciklusnál (5. Kép) a feltétel a ciklus *elején* helyezkedik el. Ez a feltétel (*F*) az *őr* szerepét játssza, hiszen amikor a ciklus elé érünk, ahhoz, hogy belépjünk a ciklustestbe, „át kell jutnunk” az őrön, azaz eleget kell tenni a feltételnek. Ha feltételre kapott válasz „igaz”, beléphetünk a ciklusba, és végighaladhatunk rajta, de csak **egyszer**. Amikor végigmentünk az úton, ismét az őrhöz (feltételhez) kerülünk, és ahhoz, hogy ismét beléphessünk a ciklustestbe, megint eleget kell tenni a feltételnek. Ha azonban nem teszünk eleget a feltételnek, a *„hamis”* ágon kell haladnunk, és ezzel kiesünk a ciklusból. Ne feledjük, a képpel ellentétben a ciklustest több tevékenységet is tartalmazhat, nem csak egyet. Az elöltesztelős ciklus érdekessége, hogy ha „a sors úgy hozza”, megtörténhet az is, hogy egyszer sem léphetünk be a ciklus belsejébe (ez akkor történik, ha rögtön a feltételre „hamis” választ kapunk).

Ezzel szemben a **hátultesztelős** ciklus (6. Kép) lehetővé teszi, hogy egyszer „*ingyen*” végighaladjunk a cikluson, ugyanis itt az őr (a feltétel) nem a ciklus elején, hanem annak *végén* van. Amikor a ciklustest végéhez érünk, az őr elé kerülünk, és eleget kell tennünk a feltételnek ahhoz, hogy ismét visszamehessünk a ciklustestbe. Itt látszódik a két típusú ciklus közötti legnagyobb különbség: míg az elöltesztelős ciklusnál megtörténhet, hogy a ciklus belsejét *egyszer sem* látogatjuk meg (ha a feltételnek már rögtön az elején nem teszünk eleget), addig a hátultesztelős ciklusnál egyszer biztosan végig fogunk a ciklustesten haladni.



5. Kép: Elöltesztelős ciklus



6. Kép: Hátultesztelős ciklus

Az elágazó struktúrához hasonlóan itt is kötelező a szerteágazó utak összefolyása, amit itt is **gyűjtő-csomópont** segítségével oldunk meg.

4. fejezet: C programozási nyelv

Ebben a fejezetben az olvasó végre megismerheti a C programozási nyelv alapjait, aminek segítségével már készíthet egyszerű programokat. A fejezet főleg elméleti szinten közelíti meg a C-vel való ismerkedést, amire egyrészt azért van szükség, mert minden programnyelv specifikus és pontosan megszabott nyelvtannal (ún. *szintaxissal*) rendelkezik, aminek elsajátítása kulcsfontosságú az adott programnyelvben való programozásnál, másrészt a programozás világában nem tűrik meg a szintaxisból eredő hibákat. A természetes nyelvekben a nyelvtan és a szókincs kulcsfontosságú ahhoz, hogy kommunikáljunk, de a programnyelvekben, ahol lényegében a számítógéppel kommunikálunk, a hús-vér beszélgetőpartnerrel ellentétben a számítógép nem fogja megtűrni a nyelvtani és szókincsből eredő hibákat. Legtöbbször már a fordítás során észreveszi a hibákat, és az egész programot hibásnak fogja tartani. Egyes hibákat viszont nem fog így sem észrevenni, hanem a program futása során fogja a felhasználó tapasztalni – ennek eredménye a program lefagyása, hirtelen kilépése, furcsa viselkedése, stb. A manapság egyre divatosabb gyakorlati nyelvi oktatás hasznos lehet, de a programnyelvek esetében talán túl kockázatos, hiszen lehet, hogy a példákon, párbeszédeken alapuló nyelvtanulás megtanítja a résztvevőt beszélni, de nem biztos, hogy helyesen. Épp ezért fontos, hogy az olvasó megismerkedjen az elméleti résszel is. Természetesen a gyakorlati rész is nagyon fontos, így minden új információ helyes és helytelen példákkal lesz alátámasztva.

E fejezetben az olvasó először pár adatot fog arról megtudni, hogyan alakult ki a C programozási nyelv, milyen programnyelvről van szó, és megtudja azt is, mi az a lexéma. Majd bemutatjuk az első C programunkat, pár szót szólva a program minden soráról. Ezután megindul a különböző fogalmak bővebb magyarázata: először az előfeldolgozóról és a direktívákról lesz szó, majd a megjegyzésekről (kommentárokról), majd ezeket fogják követni az azonosítók, kulcsszavak, változók, konstansok és operátorok. Ezután rá kell térni az algoritmusoknál megismert adatkiírás és adatbevitel C-beli megfelelőjének részletezésére. Ezáltal az olvasó elég tudást szerez ahhoz, hogy megírja a legelső igazi (habár még mindig szekvenciális) programjait – tehát gyakorlati feladatok fognak következni. Ezek még ki lesznek bővítve a matematikai könyvtári függvényekkel – gyakorlati szempontból is. Az ezután következő részek már kevesebb elméleti résszel rendelkeznek, így relatív hamar rá lehet térni a gyakorlati feladatokra. Először az elágazások és az arra épülő gyakorlati feladatok fognak jönni – az algoritmusstruktúrák második formája. Ezeket fogják a ciklusok követni – úgy elméletileg, mint gyakorlatilag. A ciklusok után a tömbök fognak következni, amik igen közkedveltek a programozás világában. A fejezetet a függvények fogják zárni, amik szintén igen fontosak minden programozó számára.

Pár szó a C-ről, mint programozási nyelvről

A C programnyelvet 1972-ben *Dennis Ritchie* fejlesztette ki (többek között), amikor még a *Bell Telephone Laboratories*-nél dolgozott. Habár magas szintű programnyelvről van szó, ez a szint elég alacsony, tehát nagyon közel áll a számítógép hardverjéhez. Mivel a C-nek közvetlen hozzáférése van a számítógép processzorához (többek között), ezért a C főképp rendszeralkalmazások fejlesztésére szolgál, és ez is volt a készítők célja.

A C egy *imperatív*, *strukturált*, *procedurális* programnyelv. Ez röviden összefoglalva a következőket jelenti:

- **Imperatív** – a program imperatív módon van írva, tehát parancsok, utasítások sorozatát tartalmazza;
- **Strukturált** – a problémát legtöbbször úgy oldja meg, hogy a feladatokat kisebb egységekre, részfeladatokra bontja, amiket aztán még tovább bont, és végül
- **Procedurális** – a probléma részproblémákra való bontását úgy oldja meg, hogy a részproblémákat függvényekbe (szubrutinokba) helyezi, amiket aztán csak hívni kell. Minden függvény végrehajtásakor egy eredmény születik (a részprobléma megoldása), amit aztán felhasználunk a főprobléma megoldására.

Habár mai szemmel nézve igen koros programnyelvről van szó, még mindig az egyik legnépszerűbb programnyelv rendszeralkalmazások készítésére. Azonban a C **általános célú** programozási nyelv is, mivel sok másra is használható, mint pl. tudományos, műszaki, szimulációs célokra. Habár oktatási célokra igen nehéz használni, mivel nehéz az alapokat elsajátítani (erre a célra sokkal jobb választás a Pascal, vagy a Modula-2), a műszaki középiskolák többségében mégis a tanterv része, mivel sok nyelv szintaxisa pont a C-re épül, mint a C++, C# és a Java. Ezek a nyelvek már objektumorientált programnyelvek, amik leírására most nem fogunk kitérni.

A C nyelv természetes operációs rendszere a *UNIX*, amin alapszanak a *LINUX* operációs rendszerek is. Ma minden *UNIX* és *LINUX* operációs rendszer szerves része a C, sőt, magát a *UNIX*-ot is javarészt C-ben programozták. Természetesen azóta más operációs rendszereken is megjelent, így *Windows*-on is.

Ami a C **jelkészletét**, azaz **ábécéjét** illeti, a legtöbb programnyelvhez hasonlóan itt is az angol ábécé betűiből, numerikus karakterekből (tehát számokból) és egy csomó speciális karakterből áll:

- **Angol ábécé betűi** – itt nagyon fontos kiemelni, hogy a C különbséget tesz a kis- és nagybetűk között, tehát a nagy ’A’ nem egyenértékű a kis ’a’ karakterrel. Más programnyelvek, mint pl. a Pascal nem tesz különbséget a kis- és nagybetűk között. Fontos kiemelni, hogy a C csakis az angol ábécé betűit ismeri fel, a magyar nyelv ékezetes betűit ezért mindig mellőzzük.
- **Numerikus karakterek** – a decimális (tízes) számrendszer számjegyei: 0, 1, 2, 3, 4, 5, 6, 7, 8 és 9.
- **Speciális karakterek** – ide tartoznak a következő jelek: `+ = _ - ( ) * & % # ! | . , : ; / ? < > { } ~ \ [ ] ^ ‘ “`
- **Fehér karakterek** – ide tartoznak: szóköz, tabulátor, új sor, és a megjegyzések (kommentárok). Egyrészt a szintaktikai egységek elválasztására szolgálnak, de a fölösleges

fehér karaktereket (pl. a megjegyzéseket) elveti a fordító. A megjegyzésekről hamarosan bővebben is beszélni fogunk.

Ebből az ábécéből épülnek fel a C **lexikai szimbólumai** (**lexémái**). A lexikai szimbólumok olyan szavak, kifejezések, amiknek már jelentésük van, tehát ezek képezik a C programnyelv szótárát. A lexémákat fehér jelekkel válasszuk szét egymástól, mint ahogy a természetes nyelvekben is. A C-ben a lexémák lehetnek **azonosítók** (nevek, **identifikátorok**), **konstansok**, **kulcsszavak** és **operátorok**. Természetesen a későbbiek folyamán minddel külön meg fogunk ismerkedni.

A lexikai szimbólumok segítségével épülnek fel a C további konstrukciói. Ezek a következők:

- **Parancsok (utasítások)** – segítségükkel adunk konkrét parancsokat a számítógépnek, amik végrehajtásával egyre közelebb kerülünk a probléma megoldásához;
- **Program** – maga a probléma;
- **Direktívák** – az előfeldolgozó utasításai. Nem tartoznak konkrétan a programhoz, de bizonyos utasításokat adnak magának a fordítónak. Később még lesz róluk szó.

Az első C program

Tekintsük a következő egyszerű C nyelven írt programot:

```
#include <stdio.h>

main()
{
    printf("Hello World!\n");
}
```

A program kimenete az idézőjelek között szereplő felirat, tehát:

```
Hello World!
```

Pár megjegyzés a fenti programmal kapcsolatban:

- Ahogy már említettük, a C programozási nyelv különbséget tesz a kis- és nagybetűk között. Minden C parancsnak kisbetűsnek kell lennie.
- A C program belépési pontját a `main()` függvényhívás azonosítja. Minden program a `main()` ponttól kezd el futni, ami magában is függvény, méghozzá a fő függvény. Egyelőre a függvényt argumentumok (paraméterek) nélkül hívtuk meg, a későbbiekben ennek részletezésére még kitérünk.
- A `{` és `}` a kezdő és a végpontját jelölik a végrehajtási résznek.
- `#include <stdio.h>` nélkül nem működne a `printf()` függvény. Ez egy direktíva, amire hamarosan bővebben kitérünk.
- A `printf()` egy függvény, adatkiírásra használjuk. A függvény zárójelében a kiírandó szöveget dupla idézőjelek közé kell tennünk. Vegyük észre azt is, hogy a `\n` karakter nem került kiírásra. Tehát a `printf()` külön tartalmazza a kiírandó szöveget és a kiírást befolyásoló változókat. Ami a dupla idézőjelek között megjelenik, változtatás nélkül kiírásra kerül, kivétel ez alól a `\` és `%` karaktereket követő jel, vagy jelsorozat. A `\n` karakter a fordító számára azt jelenti, hogy a következő karakter kiírása előtt új sort kell kezdenie.
- A parancsokat pontosvessző (`;`) zárja.

A továbbiakban megkezdjük a C programozási nyelv alaposabb bemutatását.

Az előfeldolgozó (Preprocessor)

Az **előfeldolgozó** (ang. ***Preprocessor***) a C fordítójának része. Feladata a forráskód felkészítő feldolgozása még a valódi fordítás előtt. Az előfeldolgozó parancsait, utasításait **direktíváknak** (ang. *Directive*) nevezzük, segítségükkel felkészítjük a szöveges forrásprogramot a fordításra. Ez a felkészítés magába foglalja a szükséges fájlok bemásolását a forrásprogramba, de segítségükkel bizonyos lexémákat is fel tudunk cserélni tetszőleges kifejezésekre (ún. makrókra).

Fontos megjegyezni, hogy az előfeldolgozó direktívái nem tartoznak a C utasításai közé, ezért írásuk is jócskán eltér. Figyeljünk oda a következő szabályokra:

- A direktívát mindig **#** (ang. *Hash*, magyarul sokszor a *„kerítés”*, vagy *„kettős kereszt”* kifejezéssel illetik) jellel kell kezdeni;
- A direktívák végére (C utasításokkal ellentétben) **tilos** pontosvesszőt (**;**) tenni;
- Minden direktívát új sorba kell írni (ez azért lényeges, mert nem teszünk végükre pontosvesszőt);
- Nem lehet egy sorba direktívát is, meg C utasítást is tenni, hiszen a direktívák az előfeldolgozóhoz, míg a C utasítások a C-hez tartoznak;
- A direktívák bárhol elhelyezkedhetnek a programban, de a leggyakoribb az az eset, amikor a program elejére tesszük őket. Minden direktíva a forrásfájl azt követő részére hat, tehát a program elejére tett direktívák az egész fájlra hatni fognak.

A *„Hello World!”* programból már láthatjuk, hogy lényegében nem tudunk valamirevaló programot írni direktívák nélkül. Habár több direktíva létezik, mi csak kettőt fogunk megemlíteni, az egyik a `#include`, másik pedig a `#define`.

Az #include direktíva

A C egy olyan programnyelv, ami önmagában igen keveset tud. Például magától még arra sem képes, hogy szöveget írjon ki a képernyőre (adatkiírás), de arra sem, hogy új adatot írjunk be a billentyűzeten keresztül (adatbevitel). De természetesen ez nem azt jelenti, hogy nekünk kell minden program írásakor ezt kitalálni, ugyanis a C készítői már gondoltak erre, és elkészítették őket helyettünk, függvények formájában, és ezeket egy könyvtárszerű struktúrába sűrítették. Ezeknek a függvényeknek egy fontos része az ún. **fejfájl** (ang. ***Header***). Ezek olyan **`.H`** kiterjesztésű fájlok, amik tartalmazzák az előbb említett függvények jellemzőit. Ahhoz, hogy használhassunk a C függvénykönyvtárából egy ilyen függvényt (pl. az adatkiírás függvényét), elég csak azt hívni. Ehhez segít a `#include` direktíva.

A **`#include`** direktíva lényegében beépíti, bemásolja a megadott szöveges fájlt a hívás helyére (tehát a direktíva helyére). Szintaxisa a következő:

```
#include <fájl_neve>
```

…ahol `fájl_neve` a bemásolandó fájl neve, `<` és `>` jelekkel körülvéve. A direktíva megkeresi a C beépített függvénykönyvtárában ezt a fájlt, és bemásolja a direktíva helyére. A *„Hello World!”* programnál láttuk már ezt a direktívát, méghozzá így: `#include <stdio.h>`. Ez a direktíva

beolvassa a `stdio.h` nevű fejfájlt és bemásolja a program elejére. Ez a fejfájl tartalmazza az alapvető ki- és bemeneti utasításokat, tehát az adatkiírást és az adatbevitelt (ang. ***ST****andar****D*** ***I****nput* ***O****utput*, jelentése „standard bemenetel kimenetel"). A *„Hello World!"* programban használtuk is az egyiket, ez a `printf()` függvény, ami adatkiírásra szolgál. Az adatkiírásról és adatbevitelről később részletesen fogunk beszélni.

Természetesen a C függvénykönyvtára nem csak a `stdio.h`-ból áll, hanem még nagyon sokból. Használni fogunk még más fájlokat is, mint pl. a `stdlib.h` és `math.h` fájlokat.

A #define direktíva

A **`#define`** direktíva ún. **makrók** létrehozására szolgál. Lényegében egy bizonyos szöveget lecserél egy másikra, hasonlóképp, mint a *Microsoft Word*-ben. Ezzel konstansokat hozhatunk létre. Így például ha tudjuk, hogy a π értéke 3.14, akkor ezt konstansként is definiálhatjuk, aminek nevet adunk (például `PI`). Ez lehetővé teszi, hogy a programban szabadon használjuk a `PI` kifejezést, és habár a C nem tudja, hogy mi ez a kifejezés, az előfeldolgozó a fordítás előtt minden PI kifejezést lecserél 3.14-re.

A makrók létrehozásáról, és a `#define` direktíva helyes használatáról a *Konstansok* résznél fogunk részletesen beszélni.

Megjegyzések (kommentárok)

A **megjegyzéseket (kommentárokat)** a programszöveg magyarázatainak beírására szoktuk használni. Hibakereséskor is jó hasznát vehetjük azonban, így ha a program egy részének végrehajtását szeretnénk kihagyni, akkor azt megjegyzés blokkba zárhatjuk. Más szóval, a megjegyzések nem a számítógépnek szólnak, hanem a programozónak, és a program **dokumentálásra** szolgálnak, tehát arra, hogy mi a program feladata, hogyan működik, stb. Ezért a megjegyzéseket a **fehér karakterek** közé soroljuk.

A megjegyzés a **/*** karakterpárossal kezdődik, és a ***/** karakterpárossal végződik, azaz szintaxisa a következő:

```
/* megjegyzés szövege */
```

A megjegyzés lehet **egysoros**, vagy **többsoros**, tehát a következő két példa teljesen legális:

```
/* ez most egy egysoros megjegyzés */

/* ez most
   több soros
   megjegyzés */
```

Ezenkívül teljesen legális az is, hogy egy sorban legyen C utasítás is, meg egy megjegyzés is (ezt akkor használjuk, amikor egy nehezen érthető utasítást szeretnénk megmagyarázni). Például:

```
printf("Hello World!\n");    /* adatkiiras */
```

Azonban nem lehet megjegyzéseket egymásba ágyazni, tehát nem lehet egy megjegyzésen belül még egyet indítani. Tehát a következő példa helytelen:

```
/* Ez itt /* egy rossz */ megjegyzes. */
```

A megjegyzések nem a számítógépnek szólnak, ezért az előfeldolgozó fordítás előtt kikeresi az egész forrásfájlban az összes megjegyzést, és kitörli őket, de persze továbbra is megmaradnak a forrásfájlban.

Kiemelnénk, hogy a C-fordító 1999-es frissítésébe (amit *C99*-nek neveznek) betették a C++ programozási nyelv újszerű kommentár-módját is, ennek jelölése **//**, és kizárólag **egysoros** megjegyzések írására szolgál. Írása egyszerű, csak ki kell tenni a kommentár elé a // jelpárost, majd írni a megjegyzést. A megjegyzést nem kell bezárni a jelpárossal, mivel az a sor végéig tart, tehát elég csak *Enter*-t ütni, a következő sor szövege már nem lesz a megjegyzés része. Tehát, a szintaxis:

```
// megjegyzés szövege
```

A // megjegyzések ma igen népszerűek, legtöbbször C utasításokkal kombinálva használják. Tehát az előző példát leírhatnánk így is:

```
printf("Hello World!\n");    // adatkiiras
```

Egyébként az előzőleg bemutatott `/*` megjegyzéseket **C-megjegyzéseknek**, míg az újszerű `//` megjegyzéseket **C++-megjegyzéseknek** is nevezik. Azonban a `//` megjegyzésekkel legyünk óvatosak, ha 1999-nél idősebb fejlesztőkörnyezetben (vagy fordítóval) dolgozunk, bajba keveredhetünk, hiszen nem lesznek felismerve.

A megjegyzéseket az egész jegyzeten keresztül használni fogjuk magyarázatok írására, azonban nem fogjuk őket annyira szigorúan venni, így például használni fogjuk benne a magyar ékezetes betűket is. Habár ezek írása sok implementációban tilos lenne, mi ezen át fogunk siklani, mivel mégis egy jegyzetről van szó, aminek célja az olvasó oktatása.

Az azonosítók és a kulcsszavak

Ahogy már említettük, a lexémák többek között lehetnek azonosítók és kulcsszavak. Az **azonosítókat** (**neveket**, **identifikátorokat**, ang. *Identifiers*) akkor használjuk, amikor változókra, szimbolikus konstansokra, adattípusokra, függvényekre, stb. szeretnénk hivatkozni. Például az algoritmusoknál szükség volt a változóknak nevet adni, pl. *x*, *y*, *árfolyam*, *fakt*, *ered*, stb. Ezek mind azonosítók. Azonban az algoritmusokkal ellentétben a C-ben már nem használhatunk ékezetes betűket, csak az angol ábécé betűit használhatjuk.

Az azonosítóra, azaz névre a következő megkötések érvényesek:

- Csak **betűket**, **számjegyeket** és **aláhúzás** karaktert (**_**) tartalmazhat;
- Betűvel kell kezdődnie (kivételes esetben aláhúzás karakterrel, de nem ajánlatos);
- Hossza legfeljebb 32 karakter lehet (bár ez implementációfüggő).

Ha visszaemlékezünk, a szintaxisdiagramnál példaként lerajzoltuk az azonosítót. A diagramot szinte változatlan formában tudnánk itt is használni, két kikötéssel: először, a betűk szintaxisdiagramjához hozzá kellene tenni az aláhúzás karakterét is, és másodszor, mivel a C különbséget tesz a kis és nagybetűk között, ezért a betűk diagramját még ki kellene bővíteni a kisbetűkkel is (a, …, z).

Érdemes megjegyezni, hogy habár engedélyezett, nem ajánlatos az aláhúzás karakterrel kezdeni az azonosítókat, mivel azokat csak a C függvénykönyvtárában szokás használni.

Engedélyezett azonosítók például: `szam`, `kezdo_betu`, `elsoErtek`, `ertek1`, `_nemAjanlatos`, stb.

Nem engedélyezett azonosítók például: `2ab` (nem kezdődhet számmal), `b-3` (nem engedélyezett karakter), `c:6` (nem engedélyezett karakter), `elso ertek` (nem tartalmazhat szóközt), stb.

Bizonyos azonosítók speciális jelentést hordoznak. Ezeket az azonosítókat **kulcsszavaknak** nevezzük.[11] Jelentésük a C-ben előre meg van határozva, és nem változtathatók. Ezeket a kulcsszavakat nem használhatjuk névként a programban, tehát nem lehet változókat, szimbolikus konstansokat, adattípusokat és függvényeket így elnevezni. Habár a C-ben elég sok kulcsszó van, elég, ha itt csak párat említünk meg (mint látjuk, mindegyiket kis betűkkel írjuk):

`break`, `case`, `char`, `const`, `continue`, `default`, `do`, `double`, `else`, `enum`, `float`, `for`, `goto`, `if`, `int`, `long`, `return`, `short`, `signed`, `sizeof`, `static`, `struct`, `switch`, `typedef`, `unsigned`, `void`, `while`

11 Bizonyos szakirodalmak kulcsszó helyett a *„foglalt szó”* kifejezést használják.

A változók

Az algoritmusoknál láthattuk, hogy a legegyszerűbb *„Hello World!”* feladaton kívül minden algoritmus használt változókat. Arról is már volt szó, hogy a programozásnál fontos tudni, melyik bemeneti adatok engedélyezettek, azaz legálisak. Itt nem csupán arról van szó, hogy egész számokat, valós (lebegőpontos) számokat, vagy betűket írhatunk be, hanem arról is, hogy például milyen **intervallumban** mozoghatnak a beírt számok (ún. **értékkészlet**). Ez miatt a programnyelvekben, így a C-ben is, bevezették az **adattípus** (vagy röviden csak **típus**, ang *Type*) fogalmát. Amikor a programban egy változót vezetünk be, annak mindig meg kell adni a típusát is: lényegében ez fogja elárulni, hogy milyen engedélyezett értékeket vehet fel ez a változó.

A C-ben minden változót, mielőtt használnánk, definiálni (deklarálni) kell. Habár nagyon sokan nem tesznek különbséget *definíció* és *deklaráció* között, a szakirodalom igen. Ennek értelmében a **deklaráció** (leírás, ang. *Declaration*) során a változónak megadjuk a nevét (azonosítót rendelünk hozzá) és a típusát. Másfelől, a **definíció** (ang. *Definition*) tartalmazza a változó deklarálását, de ezen kívül még lefoglalja a változóhoz szükséges memóriát, és lehetőség szerint még kezdőértéket is ad a változónak. A változókat mi mindig *definiálni* fogjuk, tehát rögtön lefoglaljuk a változóhoz szükséges memóriát is, de nem fogunk neki mindig kezdőértéket adni – ezt megadhatjuk később is.

A C-ben az adattípusokat két nagy csoportra oszthatjuk:

- **Alaptípusokra** (nevezik még **egyszerű típusoknak** is) – ide tartozik a *karakter* típus (`char`), az *egész* számok típusa (`int` és annak különböző fajtái), valamint a *valós* (lebegőpontos) számok típusa (`float` és `double`).
- **Származtatott** (vagy **összetett**) **típusokra** – ide tartoznak a *tömbök*, *függvények*, *mutatók*, *struktúrák* és az *uniók*. Ezekből a típusokból mi csak a tömbökkel és a függvényekkel fogunk megismerkedni.

Pár szó a típusmódosítókról (haladó szint)

Tehát, mint látjuk, az alaptípusok definiálására alapvetően négy kulcsszavat használunk: `char`, `int`, `float` és `double`. Azonban ezek elé egyéb kulcsszavakat, ún. **típusmódosítókat** is tehetünk, amivel úgymond testre szabhatjuk ezeket az alaptípusokat, új alaptípusokat kapva. Négy ilyen módosító létezik: `short`, `long`, `signed` és `unsigned`. A **`short`** leszűkíti az engedélyezett értékek intervallumát, a **`long`** pedig kibővíti azt. Ezzel szemben a **`signed`** (**előjeles**) úgy válogatja az értékintervallumot, hogy legyenek benne pozitív és negatív számok is, míg az **`unsigned`** (**előjel nélküli**) csak pozitív számokat engedélyez. Ki kell emelni, hogy nem mindegyik alaptípus engedélyezi a típusmódosítókat, és ha engedélyezi is, nem feltétlenül mind a négyet.

A karakter típus

A **karakter típus** kulcsszava a **`char`** (ang. *Character*). A `char` típusú változók **egy darab karaktert** menthetnek el a memóriába (például: ’`A`’, ’`b`’, ’`4`’, ’`!`’, stb.). A következő szabályok érvényesek rá:

- Egy karakter típusú változó értékének megadásakor a karaktert **egyszeres idézőjelbe** (aposztrófba, **’**) kell tenni, pl. **’`a`’**, **’`U`’**, **’`3`’**, stb.

A karakter típus egyéb szabályai (haladó szint)

- Egy bájtnyi (8 bit) memóriát foglal le magának, ami összesen 256 értékkombinációt (azaz karaktert) jelent. Ugyanis a számítógépekben minden karakternek megvan a maga bináris reprezentációja, ami lényegében annyit jelent, hogy a karakterek (így a betűk és a speciális karakterek is) lényegében számok. Az előbb említett 256 kombináció épp elég arra, hogy beleférjen az angol ábécé (külön a kis és a nagy betűk), a számjegyek és a speciális karakterek.
- A legtöbb implementációban alapértelmezettként előjeles (`signed`) változatban szerepel (tehát `signed char` helyett elég csak azt írni, hogy `char`). Létezik előjel nélküli változat is (`unsigned char`).
- A `char` (azaz `signed char`) értékkészlete [-128, …, 127], míg az `unsigned char` értékkészlete [0, …, 255].
- Nem támogatja a `short` és `long` módosítókat.

Az egész típusok

Az **egész számok típusának** kulcsszava az **int** (ang. *Integer*). Ez kétségkívül a leggyakrabban használt alaptípus, hiszen a legtöbb esetben egész számú változókra lesz szükségünk. A következő szabályok érvényesek:

- Az egész számok típusának két fő fajtája az **`int`** és az **`unsigned int`**. A sima `int` segítségével pozitív és negatív egész számokat is írhatunk, míg az `unsigned int` segítségével csak pozitív számokat és a nullát. Ha a matematikában használt számhalmazokat néznénk, akkor az `int` megfelelne a $\mathbb{Z}$ halmaznak, míg az `unsigned int` az $\mathbb{N}_0$ halmaznak.
- Az `int` értékkészlete implementáció- és hardverfüggő. Régen két bájtnyi (16 bit) memóriát foglalt le magának, de ma ez sokszor már négy bájt (32 bit). Ennek függvényében az értékkészlet 16 bit esetén [-32768, …, 32767] (`int`, azaz `signed int`), vagy [0, …, 65535] (`unsigned int`). 32 bit esetén pedig [-2 147 483 648, …, 2 147 483 647] (`int`), vagy [0, …, 4 294 967 295] (`unsigned int`).

Az egész típusok egyéb szabályai (haladó szint)

- Az összes módosító támogatva van, így a következő variációk lehetségesek: `signed int` (megegyezik az `int` típussal, tehát amikor előjeles egész számokat szeretnénk írni, elég csak az `int` kulcsszavat használni), `unsigned int`, `signed short int`, `signed long int` (a `signed`-et elhagyhatjuk, tehát elég csak azt írni, hogy `short int` vagy `long int`), `unsigned short int` és `unsigned long int`.
- A számokat nem csak tízes számrendszerben adhatjuk meg, hanem oktális (nyolcas) és hexadecimális (tizenhatos) számrendszerben is. Ezek írására nem fogunk külön kitérni.

A valós (lebegőpontos) típusok

A **valós számokat** a programnyelvekben hivatalosan **lebegőpontos** (ang. *Floating Point*) számoknak nevezzük. A lebegőpontos számoknak két kulcsszava van C-ben, ezek a `float` és a `double`. A `float` adattípust **egyszeres pontosságú** lebegőpontos számok tárolására használjuk, míg a `double` típust **kétszeres pontosságú** lebegőpontos számok tárolására. A következő szabályok érvényesek:

- Az értékkészlet `float`-nál kb. $\pm 10^{\pm 38}$, `double`-nél pedig kb. $\pm 10^{\pm 308}$.
- Ami a pontosságot illeti (tehát, hogy hány tizedes pontossággal írja ki a nem egész számokat), a `float`-nál ez a szám 6, míg `double`-nél a pontosság 15.
- A nagy értékeket a C nyelv *exponenciális* formában írja ki, amit `E` betűvel (exponens) jelöl. Így pl. a 3.4×10^{-8} számot úgy írja, hogy `3.4E-08`.
- Valós típusú változót csak tízes számrendszerben adhatunk meg. Amikor valós típusú változó értékét kell megadni, ezt kétféleképp tehetjük meg: használhatunk tizedespontot (ezt ponttal jelöljük, pl. `3.14`), vagy használhatjuk az előbb említett exponenciális módszert.

A valós típusok egyéb szabályai (haladó szint)

- A `float` nem támogatja a módosítókat, és lényegében előjeles (`signed`) formában tárolja a számokat. A `double` viszont egy módosítót támogat, ez a `long`, amivel még nagyobb pontosságú számokat lehet írni. A `double` is előjeles (`signed`) formában tárolja a számokat.
- A lebegőpontos számokkal eszméletlen nagy számokat lehet tárolni, és ne feledkezzünk el a tizedes számjegyekről sem. Habár implementációfüggő szokott lenni, a `float` négy bájtnyi (32 bit) memóriát foglal le, a double pedig nyolc bájtnyit (64 bit).

A logikai adattípus

Az algoritmusoknál gyakran volt elágazásoknál és ciklusoknál szükség arra, hogy a döntés-csomópontban feltegyünk valami kérdést, amire két lehetséges választ kaphattunk: igaz, vagy hamis. A programnyelveknek ez miatt fontos a **logikai adattípus** léte. Azonban a C kivételt képez ez alól, ugyanis a C-ben **nincs** direkt logikai adattípus (ennek kulcsszava legtöbbször `boolean` szokott lenni), de persze ez nem azt jelenti, hogy nem lehet azt mással helyettesíteni. C-ben a logikai adattípust az `int` típussal **helyettesítjük**, méghozzá a következőképpen: ha a változó értéke `1` (lényegében bármely szám nullán kívül), akkor ennek jelentése **igaz** (ang. *True*), ha pedig a változó értéke `0`, akkor a jelentése **hamis** (ang. *False*).

A változók definiálása

A korábbi részekben megismerkedtünk a C alaptípusaival (`char`, `int`, `float` és `double`, valamint ezek alfajtái), most pedig még az maradt hátra, hogy megnézzük, miként tudunk ilyen típusú változókat definiálni. A szintaxis leegyszerűsítve a következő:

```
típus változó_neve < = kezdőérték> <, …>;
```

…ahol `típus` a típus kulcsszava (pl. `int`), `változó_neve` a definiálandó változó azonosítója (neve), `kezdőérték` pedig a változó kezdőértéke. A változó nevét, azaz azonosítóját az előző részben taglalt törvényszerűségek szerint adjuk meg. A `<` és `>` jeleket nem írjuk, az ezekkel körbevett részek opcionális (nem kötelező) részeket képeznek. Így például látszódik, hogy nem kötelező a változónak kezdőértéket adni.

Tehát látjuk, hogy először a típust kell megadni, aztán jön a változó neve (vagy a változók nevei **vesszővel** elválasztva, ha több identikus típusú változót kell definiálni). Minden változónak tetszőlegesen **kezdőértéket** is adhatunk (a kezdőérték elé **egyenlőség** (`=`) jelet kell tenni), de ez nem kötelező ezen a helyen, ugyanis lehet később is. Azonban vigyázzunk, sokan beleesnek abba a hibába, hogy elfelejtenek a változónak értéket adni, ami hibákhoz fog vezetni a változó olvasásakor.

Külön felhívjuk a figyelmet arra, hogy a definíciós sort **pontosvesszővel** (`;`) kell lezárni.

A következő példák jól demonstrálják a változók *helyes* definiálását:

```
int alfa;                    /* int típusú változó definiálása
                                kezdőérték nélkül */
unsigned int beta = 2;       /* unsigned int típusú változó
                                definiálása kezdőértékkel */
float gamma, delta;          /* mindkét változó típusa float lesz*/
int elso = 1, masodik = 2;   /* mindkét változónak adtunk kezdőértéket */
char betu = 'E';             /* char esetén az értéket (betűt) egyszeres
                                idézőjelbe kell tenni */
```

A programunkban annyi változót definiálhatunk annyi sorban, amennyit akarunk. Ez miatt a következő két stílusú definiálás teljesen egyenértékű:

```
float gamma, delta;
```

```
float gamma;
float delta;
```

Érdemes megemlíteni még azt, hogy a változók definiálását tehetjük akárhova, de logikus módon illik a program elejére tenni.

A végén foglaljuk még egyszer össze a változókról tanultakat egy táblázatba:[12]

12 A táblázat tartalmazza a *haladó* szinten említett egyéb adattípusokat is.

Adattípus	Értékkészlet	Méret (byte)	Pontosság (jegy)
`char`	-128, …, 127	1	
`unsigned char`	0, …, 255	1	
`int`	-32768, …, 32767	2	
`unsigned int`	0, …, 65535	2	
`long int`	-2 147 483 648, …, 2 147 483 647	4	
`unsigned long int`	0, …, 4 294 967 295	4	
`float`	3.4e-38, …, 3.8e+38	4	6
`double`	1,7e-308, …, 1,7e+308	8	15
`long double`	3.4e-4932, …, 3.4e+4932	10	19

Saját típusok előállítása (haladó szint)

Mint láthattuk, egyes alaptípusok elég hosszúra sikeredtek (pl. `unsigned long int`). Ha sokszor kell ilyen típusokat használnunk, akkor talán jó ötlet lenne helyette létrehozni egy ún. **saját típust** (nevezik még **felhasználói típusnak** is). Ennek kulcsszava a **`typedef`**. Létrehozása és használata egyszerű. Először definiálni kell a felhasználó típust a következő szintaxis szerint:

```
typedef típus saját_típus_neve;
```

…ahol `típus` a létrehozandó felhasználói típus típusa, `saját_típus_neve` pedig annak az azonosítója. Aztán ha szeretnénk egy ilyen típusú változót definiálni, azt a változók definiálásánál használt szintaxis szerint tehetjük meg, csak a változó típusának kulcsszava helyett használjuk a saját típus nevét.

Például tételezzük fel, hogy egy programnál gyakran kell `unsigned long int` típusú változókat létrehozni. Az egyszerűség kedvéért ekkor használhatunk saját típust, méghozzá a következőképp:

```
typedef unsigned long int hosszuSzamok;    // saját típus készítése
hosszuSzamok szamlalo;                     // saját típus használata
```

Konstansok

Hivatalosan a C-ben minden olyan érték (szám, vagy karakter), amihez nem rendeltünk azonosítót (tehát változót), **konstansnak** számít. Így például a `2 * x` kifejezésben (ami kiszámolná az `x` változó kétszeresét) a `2` egy konstans.

Az emberi felfogásban a konstans, vagy állandó viszont egy olyan változó, aminek értéke állandó, és utólag nem változtatható meg. A C-ben (mivel egy elég koros programnyelvről van szó) direkt módon ilyen fajta konstansokat nem lehet definiálni, de persze a készítők erre is gondoltak, és létrehoztak az előfeldolgozóban egy olyan trükkös direktívát, amivel konstans értékű változókat lehet definiálni. Segítségével tehát olyan változókat definiálhatunk, amik értékei a program futása során nem fognak változni, mint például a π szám értéke, vagy bármely szám, esetleg karakter, aminek jelentősége van a programunkban. Ezeket a változókat egyszer definiáljuk, rögtön megadjuk a kezdőértéküket, és aztán ezt az értéket többet nem írhatjuk át, csak használhatjuk (olvashatjuk).

Habár a szakirodalom több módszert említ konstansok létrehozására, mi csak egyet fogunk részletezni, méghozzá a makrókat.

Makrókat, azaz **szimbolikus konstansokat** az előfeldolgozó már korábban említett **`#define`** direktívájával hozhatunk létre. Ez a direktíva úgy viselkedik, mint *Microsoft Word*-ben a *Find and Replace* (*Keresés és csere*) parancs: megkeresi a beírt kifejezés összes előfordulását a szövegben, majd lecseréli a találatokat a másik kifejezésre. A direktíva szintaxisa a következő:

```
#define makrónév helyettesítő_szöveg
```

…ahol `makrónév` az a kifejezés, *amit* le szeretnénk cserélni, míg `helyettesítő_szöveg` az a kifejezés, *amire* szeretnénk lecserélni a talált kifejezéseket. Közös megegyezés alapján az szimbolikus konstans azonosítóit CSUPA NAGY BETŰKKEL írjuk, hogy a programunkban könnyen megkülönböztessük a többi változótól. A többi direktívához hasonlóan itt sem szabad a direktíva végére pontosvesszőt tenni. A direktívát a program elejére szokás tenni rögtön a `#include` direktívák alá.

Tehát, mint látjuk, lényegében szó sincs arról, hogy konstans értékű változót hoztunk volna létre, csupán egy trükköt használ az előfeldolgozó: a makró nevét kicseréli valamilyen szövegre. Például tételezzük fel, hogy szeretnénk a programunkban a π szám nyolc tizedesig terjedő változatát használni. Ekkor a következőképp írhatjuk fel a direktívát:

```
#define PI 3.14159265
```

…ahol a `PI` az a kifejezés, *amit* le szeretnénk cserélni, míg a `3.14159265` az a kifejezés, *amire* szeretnénk cserélni.

A makrók használata egyszerű. Minden helyre, ahol szeretnénk használni, csak le kell írni a makró nevét, és az előfeldolgozó fordítás előtt lecseréli őket a meghatározott szövegre, így például mindenhol, ahol rátalál a `PI` kifejezésre, lecseréli azt `3.14159265`-re:[13]

13 Az alábbi programszelet utolsó három sora értékadó operátort (egyenlőség jelet) használ, amiket a következő részben fogjuk részletesen bemutatni. Érdemes lenne az értékadó operátor áttanulmányozása után még egyszer visszatérni erre a programrészre is.

```
double terulet, kerulet, r;
r = 4;
terulet = r * r * PI;    // a PI makró lecserélődik 3.14159265-re
kerulet = 2 * r * PI;    // itt is
```

A `#define` direktívának ez a legegyszerűbb használati módja. Habár még léteznek paraméterezett makrók is, ezekre nem fogunk kitérni.

A konstansok (azaz makrók) nagyon népszerűek, főleg a hosszabb, bonyolultabb programoknál, ahol egy-egy konstanst akár több tízszer, vagy százszor kell majd a programon belül használni. Két okot tudnánk felsorolni:

- **Áttekinthetőbbé és egyszerűbbé teszik a programot** – ezt láttuk már akcióban: a `PI` makró a π szám nyolc tizedesig terjedő változatát használja. Egy kicsit időigényes lenne minden alkalommal, amikor a π számot kellene használni, leírni mind a nyolc tizedest. Nem beszélve arról, hogy fennáll a félreütés veszélye (elrontjuk az egyik számjegyet).
- **Könnyebbé teszik a program későbbi módosítását** – ha későbbiek folyamán meg kell egy sokszor használt konstans értékét változtatni, nem kell az összes helyen végrehajtani a módosítást, hanem elég csak egy helyen – a `#define` direktívánál. Habár nincs nagy valószínűsége, hogy épp a `PI` makró értékét szeretnénk átírni (esetleg kibővíteni pl. 15 tizedesre), de más makróknál fennállhat a lehetőség.

A C programnyelv konstansok létrehozására a makrókat ajánlja (tehát a `#define` direktívát), mivel az így létrehozott konstansok értéke a program futása közben semmilyen módon nem változtatható meg.[14]

14 Habár a szakirodalom sokszor említ más módszereket is, főleg a `const` kulcsszó használatát, ezt a módszert nem nagyon szabad C-ben konstansok létrehozására használni, mivel elsősorban az így létrehozott konstansok hivatalosan a C definíció szerint nem is konstansok, hanem változók (ez nagyon implementációfüggő), másodsorban pedig használata nagyon korlátozott. Ez miatt mi megelégszünk a `#define` direktívával.

Operátorok

A *„Hello World”* programnál észrevehettük, hogy szinte minden sor (az előfeldolgozó utasításait leszámítva) **pontosvesszővel** ért véget. Ez nem véletlen, hiszen már említettük, hogy a lexémák segítségével további konstrukciókat hozhatunk létre, melyek között szerepelnek az **utasítások (parancsok)**. Minden parancs konkrét feladat végrehajtására kényszeríti a számítógépet, amikkel egyre közelebb kerülünk a probléma megoldásához (ez maga a program). C-ben az utasítások végére **pontosvesszőt** (**`;`** jelet) teszünk. Habár az áttekinthetőség érdekében az utasítások többségét új sorba szokás írni, ez nem kötelező, amennyiben minden egyes utasítás mögé kitesszük a pontosvesszőt.

A legegyszerűbb utasítás az **üres utasítás**, ami valójában csak egy pontosvesszőből áll:

```
;
```

Azonban ennek nincs nagy értelme, ezért az esetek túlnyomó többségében az utasítás valami konkrét feladatot tartalmaz, ezt nevezzük **kifejezésnek**. Tehát, az utasítás legtöbbször a következő formát fogja felvenni:

```
kifejezés;
```

Minden kifejezés egy vagy több lexémából épül fel, és mint tudjuk, a lexémák lehetnek azonosítók, konstansok, kulcsszavak és operátorok. Ezek közül már mindegyikről beszéltünk, kivéve az operátorokról.

Az **operátorok** alapjában véve **műveleteket** jelölnek, amikkel konkrét adatfeldolgozás történik. Minden műveletnek két alkotóeleme van: maga az **operátor**, tehát a művelet (például az összeadás művelete), valamint egy vagy több **operandus** (amiken hajtjuk végre a műveletet). Az operandusok lehetnek azonosítók és konstansok.

Az operátorokat többféle szempont szerint lehet **csoportosítani**:

- **Az operandusok száma szerint** – C-ben az operátor egy, kettő, vagy három operandust követelhet. Ez alapján léteznek **egyoperandusú**, **kétoperandusú** és **háromoperandusú** operátorok. Mint észre fogjuk venni, az operátorok többsége kétoperandusú.
- **Az operátor típusa szerint** – léteznek értékadó, aritmetikai, léptető, összehasonlító és logikai, bitműveletes, feltételes, mutató (ang. *Pointer*) operátorok, de sokak szerint a függvényhívás is egy operátor.
- **Az operátor helye szerint** – lehet **előrevetett (prefix)** és **hátravetett (postfix)**, attól függően, hogy az operátor megelőzi-e az operandust (prefix), vagy nem (postfix). Az egyoperandusú operátoroknál játszanak fontos szerepet.

Itt az operátorokat a típusuk szerint fogjuk tárgyalni, de említést teszünk a másik két szempontról is. Előtte viszont beszélnünk kell az operátorok elsőbbségéről és asszociativitásáról.

Az operátorok elsőbbsége és asszociativitása

Ez a témakör bonyolultsága miatt két részben lesz bemutatva: alapszinten és haladó szinten. Ajánlatos, hogy az alapszintet mindenki olvassa el, míg a második szintet elég, ha csak a téma iránt érdeklődők olvassák el.

Alapszinten

Egy kifejezésben több operátor is részt vehet. Ekkor viszont el kell dönteni, milyen sorrendben fogjuk őket végrehajtani. Hasonló problematikával találkozunk a matematikában is, hiszen ott is például a szorzásnak és az osztásnak elsőbbsége van az összeadással és a kivonással szemben. A programnyelvekben (így a C-ben is) az operátorok végrehajtásának sorrendje kulcsfontosságú a kétértelműségek és félreértelmezések elkerülése érdekében.

Nem minden operátornak ugyanaz az **elsőbbsége**. Így például – a matematikai törvényszerűségekkel összhangban – a szorzásnak és az osztásnak valóban *elsőbbsége van* az összeadással és a kivonással szemben. Fontos még azt is megjegyezni, hogy a C-ben az értékadó operátornak (tehát az egyenlőség jelnek, `=`) van a legkisebb elsőbbsége. Azonban a C-ben ezeken kívül még rengeteg operátor van, ezért létezik egy ún. *elsőbbségi táblázat*, amit azonban igen nehéz megjegyezni[15]. Szerencsére van egy eszköz, amivel kihathatunk az elsőbbségi sorrendre. Ezek a **kerek zárójelek `( )`**. A matematikában is ezekkel lehet kihatni az elsőbbségi sorrendre, és ez itt sincs másképp. Tehát, ha nem vagyunk biztosak abban, milyen elsőbbséggel rendelkezik valamely operátor, hangsúlyozzuk ki azt zárójelekkel.

Másik probléma az, hogy mit tegyünk akkor, amikor két különböző operátor egy elsőbbségi kategóriába tartozik (pl. a szorzás és az osztás művelete valóban egy kategóriába tartozik). Ez miatt az előbb említett *elsőbbségi táblázatot* kibővítik még egy oszloppal, ez az ún. **asszociativitás**. Ez határozza meg, milyen sorrendben kell az operátorokat végrehajtani, ha egy elsőbbségi kategóriába tartoznak. Az asszociativitás kétféle lehet: **balról jobbra** (vagyis a műveleteket balról haladva hajtjuk végre), vagy **jobbról balra**. Érdemes megjegyezni a legfontosabb műveletek asszociativitását:

- **Összeadás (+), kivonás (-), szorzás (*) és osztás (/) művelete** – az asszociativitás **balról jobbra** halad, tehát a műveleteket balról haladva hatjuk végre.
- **Értékadó operátor (=)** – az asszociativitás **jobbról balra** halad, tehát jobbról haladva hajtjuk végre.

Ha nem vagyunk biztosak, itt is kihathatunk a helyes végrehajtási sorrendre a kerek zárójelekkel.

Haladó szinten

Az **elsőbbséget** hivatalosan **precedenciának** is nevezik. Most nézzük meg az előbb említett **elsőbbségi (precedenciai) táblázatot**:

Operátor	Asszociativitás
`! ~ - ++ -- & * (típus)`	Jobbról balra
`* / %`	Balról jobbra
`+ -`	Balról jobbra

15 Az érdeklődők megtalálhatják ezt a táblázatot a *„Haladó szinten”* részben.

Operátor	Asszociativitás
<< >>	Balról jobbra
< <= > >=	Balról jobbra
== !=	Balról jobbra
&	Balról jobbra
^	Balról jobbra
|	Balról jobbra
&&	Balról jobbra
||	Balról jobbra
?:	Jobbról balra
= += -= *= /= %= <<= >>= &= |= ^=	Jobbról balra

A táblázatot a következőképpen kell olvasni: az első sor tartalmazza a legnagyobb precedenciával (azaz elsőbbséggel) rendelkező operátorokat, aztán jön a második sor, a legalsó sor pedig a legkisebb precedenciával rendelkező operátorokat foglalja magába. Ha egy kategóriába (sorba) több operátor is tartozik, azok helyközzel vannak szétválasztva, ebben az esetben az operátorok elsőbbsége azonos, hiszen egy kategóriába tartoznak. Egyes operátoroknak több jelentésük is van, ezért több helyen is szerepelhetnek.

Habár már megemlítettük az alapszintnél, a kifejezésben szereplő operátorok végrehajtásának sorrendjét két szabály határozza meg:

- **Elsőbbségi (precedencia) szabály** – akkor használjuk, amikor az operátorok **különböző** elsőbbségi kategóriákba tartoznak. Ekkor mindig az az operátor élvez elsőbbséget, ami magasabb precedenciájú kategóriába tartozik (lényegében feljebb van a táblázatban). Látjuk, hogy pl. a szorzás (*) operátora elsőbbséget élvez az összeadás (+) operátorral szemben.
- **Asszociativitási szabály** – akkor használjuk, amikor az operátorok **azonos** elsőbbségi kategóriába tartoznak (például szorzás és osztás). Ekkor a táblázat második oszlopát olvassuk, ami lényegében azt árulja el, milyen irányba haladva kell az operátorokat elvégezni, ami lehet **balról jobbra** és **jobbról balra**.

Még egyszer megemlítjük, ha nem tudjuk a táblázatot megjegyezni, inkább használjuk a kerek zárójeleket, mintsem rossz eredményeket kapjunk. Megjegyeznénk, hogy a táblázat jó pár olyan operátort is tartalmaz, amikkel most találkozunk először. Ezek közül nem fogjuk mindegyiket mélyen átvenni.

Aritmetikai operátorok

Az **aritmetikai műveletek** közé tartoznak a már rengetegszer használt alapvető műveletek: **összeadás** (**+**), **kivonás** (**-**), **szorzás** (*****) és **osztás** (**/**), de ehhez hozzá kell még venni a **maradékképzés** (egész számú maradékos osztás, modulus) műveletét is, aminek jele **%**.

Mind az öt operátor bináris, azaz kétoperandusú, kivételt képez a mínusz jel, ami lehet egyoperandusú is (erre hamarosan kitérünk).

Ami az operandusok típusát illeti, általában véve lehetnek egész, vagy lebegőpontos típusok, sőt, akár keverhetők is (pl. az első operandus egész, a másik lebegőpontos). Viszont a típusok keverése egy operátoron belül mégsem ajánlatos, de erre még kitérünk. Pár dologra ügyelni kell:

- Ha az osztás (`/`) művelet operandusai egész számok (`int`), akkor **egész számú osztásról** beszélünk, ami annyit jelent, hogy ha az eredmény nem egész szám, akkor a számítógép egyszerűen **levágja** a tizedesvessző utáni számokat. Vigyázat: a számok levágása nem azonos a kerekítéssel. Tehát, `1 / 3` eredménye `0` lesz, azonban `8 / 3` eredménye `2` lesz.
- A maradékképzés (modulus, egész számú maradékos osztás) mindkét operandusa egész típusú szám kell, hogy legyen. Például `8 % 3` eredménye `1`, mivel `8 / 3` egyenlő `2`, maradék pedig `2`, viszont `6 % 3` egyenlő `0`, hiszen nincs maradék.
- Habár matematikában gyakran elhagyják a szorzás műveletét (pl. $3x$), itt ez tilos. Vagyis ki kell rendesen írni, hogy `3 * x`.
- Szintén a matematikában gyakran szeretjük a törtjelet használni, itt viszont mindent egy sorba kell írni (segítségül használjunk zárójeleket). Így például:

Matematikában	C-ben
$\frac{a+\frac{b}{c}}{3\cdot(x+y)}$	`(a + b / c) / (3 * (x + y))`

Mint említettük, a `-` operátor egyoperandusú is lehet, ekkor nem kivonást jelent, hanem **negációt**, tehát segítségével ellentétes előjelűvé változtatható a mögötte álló operandus.

Értékadó operátorok

Még a változók definiálásánál láthattuk, hogy a változóknak rögtön adhatunk kezdőértéket is az egyenlőség (`=`) jellel, de ez nem azt jelenti, hogy később nem adhatunk nekik más értéket. A leggyakoribb operátor a C-ben kétségkívül az **értékadó operátor**, vagyis az **egyenlőség jel** (`=`). Szintaxisa a következő:

```
azonosító = érték;
```

Innen már látjuk, miért kellett annyira odafigyelni az algoritmusoknál arra, mit teszünk az egyenlőség bal és jobb felére. Ugyanis a bal felén mindig valamilyen azonosító (lényegében változó, tehát annak neve) áll, a jobb felén pedig az az érték, amit hozzá fogunk ehhez a változóhoz rendelni. Ez miatt sok programnyelv (így a C is) külön nevet ad ezeknek a kifejezéseknek, ezek a **balérték** (az egyenlőség jel bal fele) és a **jobbérték** (az egyenlőség jel jobb fele). Nézzünk meg egy programrészt pár példával:

```
int a, b;

a = 6;      /* helyes írási mód, a változó értéke 6 lesz */

9 = b;      /* rossz írási mód, a balérték nem azonosító */

b = a;      /* a jobbérték (aminek eredménye 6, mivel az 'a' változó értéke 6)
               hozzárendelődik a balértékhez, tehát b értéke 6 lesz */
```

Az = operátor jobbértéke egész kifejezés is lehet, egy vagy több operátorral. Ez miatt nagyon fontos megérteni, mi is történik a háttérben. Mint tudjuk, az = operátornak a *legkisebb* az elsőbbsége (az elsőbbségi táblázat alján szerepel), asszociativitása *jobbról balra* halad. Tehát, először a számítógép kiszámolja az = operátor jobbértékét, lényegtelen, hány operátort (pl. összeadást, szorzást, stb.) tartalmaz (vigyázzunk az elsőbbség és asszociativitás szabályára), majd a kapott eredményt hozzárendeli a balértékhez (változóhoz). Ugyanez történik pl. az előző programrészlet második sorában is (`a = 6;`), csak ott a jobbérték egy egyszerű konstans (`6`), nem tartalmaz más operátort. Fontos megjegyezni, hogy az = bal felén található változó (azonosító) régi értéke az új érték hozzáadása után végérvényesen elveszik (tehát felülíródik). Így ha az `'a'` változó tartalmazott is valamilyen más számot az értékadás előtt, az az értékadás után el fog veszni.

Nézzünk még meg egy programszeletet pár példával:

```
int c, d, e;

c = 5;

d = -c;          /* a jobbérték a c változó negációja, vagyis -5, így a d
                    értéke az értékadás után -5 lesz */

d = c + 1;       /* jobbérték eredménye 6, így a d változó értéke 6 lesz.
                    A d változó régi értéke (ami -5 volt) örökké elveszik. */

e = c * d / 2;   /* az asszociativitás szabálya szerint először a
                    szorzás, majd az osztás lesz végrehajtva, tehát
                    (c * d) / 2, ami 15, tehát az e változó értéke 15 lesz */

e = e + 10;      /* itt a bal- és jobbértéknél is szerepel az e változó.
                    A jobbérték eredménye 25 lesz (mivel az e eddigi
                    értéke 15), és az új eredmény hozzárendelődik a
                    balértékhez, tehát az e új értéke 25 lesz */

c + 1 = d;       /* helytelen, a balérték egy változó lehet, ott
                    nem történhet számolás */
```

Az aritmetikai operátorokat gyakran használjuk **akkumulálásra**, tehát a változó saját magát használja fel operandusként, azaz saját maga értékét változtatja (pl. az utóbbi programszelet utolsó előtti példája: `e = e + 10;`). Mivel az akkumuláció nagyon gyakori a programozásban, ezért a C-ben külön operátorokat találtak ki az akkumuláció tömörebb írására. Nagyon sok operandusnak találtak ki **tömör** formát. Ezek a következők: **+=**, **-=**, ***=**, **/=**, **%=**, **<<=**, **>>=**, **&=**, **|=** és **^=**, de ezek közül mi csak az első ötöt fogjuk használni. Az alábbi táblázat összesíti a leggyakrabban használt **akkumulációs (azaz tömör) értékadó operátorok** használatát:

Hagyományos forma	Tömör forma
`a = a + b;`	`a += b;`
`a = a – b;`	`a -= b;`
`a = a * b;`	`a *= b;`
`a = a / b;`	`a /= b;`
`a = a % b;`	`a %= b;`

Tehát, az előző programrészben az `e = e + 10;` helyett írhattunk volna röviden azt is, hogy `e += 10;`.

A táblázatban szereplő `'a'` mindig valamilyen változó, míg a `'b'` lehet változó, konstans, vagy akár egész kifejezés is. Így például a `g *= m - 2;` megfelel a `g = g * (m - 2);` kifejezésnek.

A tömör formájú akkumulációs operátorok nem csak rövidebbek a hagyományos írástól, hanem valamennyivel gyorsabb kódot is eredményeznek, ezért használatuk ajánlatos.

Az értékadó operátor halmozása egy utasításon belül (haladó szint)

Az előző példák igen egyszerűk voltak, hiszen csak egy darab értékadó operátort tartalmaztak. Azonban fel lehet tenni a kérdést, akkor miért van neki jobbról balra haladó asszociativitása, amit kétségkívül akkor használunk, amikor több azonos elsőbbségű operátor található egy utasításon belül. A válasz igen egyszerű: mert több értékadó operátor is előfordulhat egy utasításon belül. Azonban ez egy igen haladó szintű témakör, aminek ezen a szinten sok értelme nincs. Viszont ennek legegyszerűbb formáját felhasználhatjuk kicsit elegánsabb programok írására. Tételezzük fel, hogy egy programon belül több változónak ugyanazt az értéket kell adni. Ekkor ezt írhatjuk hagyományosan, vagy a következőképp:

Hagyományos módon	Az értékadó operátor halmozásával

`a = 7;`
`b = 7;`
`c = 7;`
	`a = b = c = 7;`

Itt már látszik, miért oly fontos az `=` operátor *jobbról balra* haladó asszociativitása. A kiértékelés tehát a következőképp megy végbe: először a szélső jobboldali értékadás hajtódik végre (`c = 7`), vagyis `c` értéke `7` lesz. Ezután a megmaradt utasítás így néz ki: `a = b = 7;`. Megint a szélső jobb `=` operátor hajtódik végre (`b = 7`), és így tovább, míg el nem érünk az `a = 7;` formáig. Ezzel az egy sorral tehát három változónak sikerült értéket adni. Ha zárójelekkel szerettük volna kihangsúlyozni az elsőbbséget, akkor így írtuk volna: `a = (b = (c = 7));`.

Léptető operátorok

A **léptető operátorok** feladata *eggyel* növelni (*inkrementálni*), vagy csökkenteni (*dekrementálni*) a változó értékét. Tehát két léptető operátor létezik: a **`++`** (inkrementálás) és **`--`** (dekrementálás).[16] Operandusként használhatunk egész, de lebegőpontos típust is.

Fel lehet tenni a kérdést, miért van rájuk szükség, hiszen az eggyel való növelés vagy csökkentés könnyen megoldható sima aritmetikai művelettel is. A válasz abban rejlik, hogy ezeket a műveleteket nagyon sokszor használjuk (főleg ciklusoknál). Sokszor megesik, hogy ilyet látunk: `i = i + 1;`. Ezt le lehet cserélni léptető operátorra is, méghozzá így: `i++;`. Hasonlóképp, `j = j - 1;` helyett írhatjuk azt is, hogy `j--;`. Azon kívül, hogy így tömörebb az írás, a számítógép sokkal gyorsabban végrehajtja az utasítást, ami fontos lehet a ciklusokban (amiket akár több százszor is végre kell hajtani). Ezért használatuk ajánlott.

16 A léptető operátorokat más szakirodalmakban az aritmetikai operátorok közé sorolják.

A léptető operátorok prefix és postfix alakja (haladó szint)

A léptető operátorokat **prefix** és **postfix** alakban is használhatjuk. Tehát mindegy, hogy azt írjuk, hogy `a++;`, vagy `++a;`. Ez azonban csak addig érvényes, amíg a parancs egyszerű, azaz nem tartalmaz a léptető operátoron kívül semmilyen más operátort (még `=` operátort sem). Ha a parancs több operátort tartalmaz (tehát összetett), akkor viszont vigyáznunk kell:

- Ha a **prefix** verziót használjuk (pl. `++a`), akkor először megnöveljük eggyel az `'a'` változó értékét, majd használjuk a kifejezésben.
- Ha a **postfix** verziót használjuk (pl. a++), akkor először az `'a'`-t használjuk a kifejezésben, majd megnöveljük az értékét eggyel.

Nézzük meg a következő programrészletet:

```
int a = 4, x, y;

x = ++a;     /* először megnöveljük az 'a' értékét, ami most 5 lesz, így
                ez lesz az 'x' értéke is. Lényegében írhattuk volna ezt
                is: a = a + 1; x = a; */

y = a++;     /* az 'a' pillanatnyi értéke 5, ez lesz az 'y' értéke is,
                majd megnöveljük az 'a' értékét, és így 6 lesz. Tehát,
                írhattuk volna ezt is: y = a; a = a + 1; */
```

Mint látjuk, lényeges a különbség a két verzió között. Sajnos vannak esetek, amikor nagyon nehéz meghatározni a helyes végrehajtást, vagy implementációtól függő. Ez főleg akkor történik meg, amikor más operátorokat is használunk a parancsban az `=` operátoron kívül, ezért legjobb őket izolálni, azaz különválasztani.

Összehasonlító és logikai operátorok

Mint már tudjuk, a C nyelvben nem létezik külön logikai adattípus, de persze szükség van rá (ezért az egész típust, tehát az `int`-et használjuk helyettesítőként, aminek értéke hamis esetén `0` lesz, igaz esetén pedig `1`).

Elágazások és ciklusok döntés-csomópontjaiban gyakran kell összehasonlítani különböző értékeket, ennek elvégzésére az **összehasonlító operátorokat** használjuk. Ezek a következők: **kisebb** (`<`), **nagyobb** (`>`), **kisebb vagy egyenlő** (`<=`), **nagyobb vagy egyenlő** (`>=`), **egyenlő** (`==`, vigyázzunk, két darab egyenlőségjellel jelöljük) és **nem egyenlő**, azaz **különböző** (`!=`). Mivel kétoperandusú operátorokról van szó, ha ezekkel összehasonlítunk két változót vagy kifejezést, akkor az eredmény `int` típusú lesz, méghozzá `1`, ha a reláció **igaz**, illetve `0`, ha **hamis**.

Például, ha egy olyan kifejezést szeretnénk írni, aminek értéke akkor legyen igaz, ha az `x` változó értéke nem negatív, akkor ezt így írhatnánk le: `x >= 0`.

Viszont most tételezzük fel, hogy azt szeretnénk kivizsgálni, hogy vajon x értéke -5 és 5 közé esik-e. Matematikában ez egyszerű: felírjuk, hogy $-5 < x < 5$. Azonban C-ben ilyet nem írhatunk, mivel tilos egy feltételbe több összehasonlító operátort tenni. Helyette szét kell bontani az összetett feltételt két egyszerűre, és azokat logikai operátorral kell összekötni. Tehát: $-5 < x$ ÉS $x < 5$.

A C-ben három **logikai operátor** létezik: **logikai ÉS** (**&&**), **logikai VAGY** (**||**) és **logikai TAGADÁS**, azaz **NEM** (**!**). Ebből az első kettő kétoperandusú, míg az utolsó egyoperandusú. Tehát az előző példát C-ben úgy írtuk volna fel, hogy `-5 < x && x < 5`. Ha úgy jobban tetszik, kitehetjük a zárójeleket a jobb kiemelésért, tehát `(-5 < x) && (x < 5)`, de nem kötelező, mivel a logikai operátoroknak kisebb az elsőbbsége. A logikai operátorok igazságtáblája matematikából már sok olvasónak ismert lehet, ha még nem, akkor íme:

a	!a
0	1
1	0

Logikai NEM művelet

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

Logikai ÉS művelet

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

Logikai VAGY művelet

Bitműveletes operátorok – pár szóban

Segítségükkel különböző bitenkénti műveleteket hajthatunk végre. Akkor használjuk, amikor alacsony szintű (ún. *assembly*) műveleteket kell biteken (bináris számokon) végrehajtani. Például a számítógép hardverelemeinek programozásánál lehetnek hasznosak. Hat operátor tartozik ebbe a csoportba: **~**, **&**, **|**, **^**, **<<** és **>>**, és nem keverendők össze az összehasonlító és logikai operátorokkal. Nem fogunk velük bővebben foglalkozni.

Feltételes operátor

A **feltételes operátor** (**? :**) az egyedüli *háromoperandusú* operátor a C-ben. Két operátorból áll: a kérdőjel elé kell tenni az első operandust, a kérdőjel és kettőspont közé a másodikat, a kettőspont mögé pedig a harmadikat.

Mivel a feltételes operátorral lényegében az egyszerűbb `if` elágazásokat lehet helyettesíteni, ezért a részletezésére majd az elágazásoknál fogunk kitérni.

Mutató (pointer) operátorok – pár szóban

A **mutató (ang. *Pointer*)** operátorokkal lehetőségünk van a memória „mély” hozzáférésére. A számítógép memóriája rengeteg kis „rekeszre”, cellára oszlik, és mindegyiknek megvan a címe. Ezekkel az operátorokkal lehetőségünk van arra, hogy megtudjuk egy változó igazi címét a memóriában, és hasonlóképp lehetőségünk van arra is, hogy memóriához férkőzzünk hozzá cím megadásával. Két operátor tartozik ebbe a csoportba: ***** és **&** (ne keverjük a bitműveletes **&** operátorral, hiszen az kétoperandusú, ez pedig egyoperandusú). Nem fogunk velük bővebben foglalkozni, viszont imitt-amott fogunk velük találkozni, így például a `scanf()` (adatbevitel) függvénynél, ami nem működne nélkülük.

Függvényhívó operátor

A függvényekről később részletesen fogunk beszélni, elég most annyit tudnunk róluk, hogy azok olyan programrészek (ún. alprogramok), amiket gyakran használunk, ezért kiemeljük őket és nevet adunk nekik. Ezután ezeket bármikor felhasználhatjuk a programjainkban, csak *hívni* kell őket. A függvény hívása után a legtöbb függvény valamilyen eredményt ad vissza.

Ahhoz, hogy egy programban egy függvényt hívhassunk, a **függvényhívó operátort** használjuk[17], aminek jelölése `()`. Kétoperandusú operátorról van szó, ahol az egyik operandus a függvény neve és a zárójel elé tesszük, a másik operandus pedig a paraméterek listája, amit a zárójel belsejébe tesszük. Ha a függvény több paraméterrel rendelkezik, azokat hívás közben vesszővel (,) választjuk el egymástól (a zárójel belsejében). Vagyis a szintaxis a következő:

```
függvénynév(paraméter1, paraméter2, …)
```

…ahol `függvénynév` alatt adjuk meg a függvény nevét, majd zárójelet nyitunk, és a zárójel belsejébe annyi paramétert írunk **vesszővel** elválasztva, amennyit definiál maga a függvény (ne felejtsük el a végén a zárójelet bezárni). Vigyázzunk, vannak olyan függvények, amik nem rendelkeznek paraméterrel, viszont a zárójelet ekkor is ki kell tenni, csak üresen hagyjuk, tehát ekkor a szintaxis:

```
függvénynév()
```

Habár a függvények részletezésére csak később fogunk rátérni, függvényhívásokkal nemsokára találkozni fogunk (*adatbevitel és adatkiírás*).

Típuskonverziók

A típuskonverzió ismét egy olyan témakört képez, ami igen bonyolult, ezért két szinten lesz elmagyarázva. Az alapszint elolvasása mindenkinek ajánlott.

Alapszinten

Bizonyos operátorfajtáknál (például az aritmetikai operátoroknál) feltehettük magunkban a kérdést, mi történik akkor, ha az operandusok eltérő típussal rendelkeznek. Például mi történik akkor, amikor a `+` operátor egyik operandusa egész szám (pl. `int`), másik viszont lebegőpontos (pl. `float`). Nagyon sok programnyelvben (így a C-ben is) a művelet elvégzéséhez fontos, hogy mindkét operandus azonos típusú legyen. Ettől függetlenül engedélyezett az operandusok típusainak keverése, viszont a fordítónak azonos típusra kell őket alakítania. Ezt nevezzük **típuskonverziónak**.

A típuskonverziók egy része automatikusan végrehajtódik, másik része viszont nem. Ezért jó pár szabályt meg kell jegyezni ahhoz, hogy biztosak lehessünk benne, hogy a program hiba nélkül fog futni. Azonban egy kis ügyességgel elérhetjük azt is, hogy típuskonverzió nélkül megússzuk a programok írását. Egyszerűen arra kell ügyelni, hogy a program írása közben minden aritmetikai operátor összes operandusa azonos típusú legyen (például legyen mindkét operandus típusa `int`). Tehát csak jól meg kell gondolni, milyen típust adunk a programban bevezetett változóknak.

17 Érdemes megemlíteni, hogy sok szakirodalomban a függvények hívása nem számít operátornak.

Haladó szinten

Ahogy már az alapszinten említettük, a típuskonverziók egy része automatikusan, a programozó beavatkozása nélkül megy végbe. Ezeket **implicit** vagy **automatikus konverzióknak** hívjuk. Ezek akkor hajtódnak végre, amikor egy kétoperandusú operátor operandusai különböző típussal rendelkeznek. Ekkor általános szabály szerint a *kisebb pontossággal* rendelkező (ún. *szűkebb*) típus automatikusan (információveszteség nélkül) átalakul a „nagyobb” (ún. *szélesebb*) típusra, és az eredmény úgy lesz kiszámolva. Viszont ha ezt az eredményt egy szűkebb típusú változóhoz kell rendelni, akkor információveszteség léphet fel, mivel az eredményt vissza kell konvertálni a szűkebb típusra. Például:

```
int i, j;
float f, m;

m = i + f;    /* az i float-ra konvertálódik (mivel f típusa float), és mivel
                 az m szintén float, ezért nincs információvesztés */

j = i + f;    /* az i float-ra konvertálódik (mivel f típusa float), így az
                 összeg is, viszont a balérték típusa most int, ami szűkebb a
                 float-nál, ezért az összeg törtrésze elveszik */
```

A programozó viszont explicit módon is beleavatkozhat a típuskonverzióba, méghozzá a **típuskonverziós operátor** (ang. ***Cast***) segítségével. Mivel itt a konverzió a programozó kérésére történik, ezért ezt **explicit típuskonverziónak** nevezzük.

A *cast* operátor egyoperandusú, és úgy használjuk, hogy a kifejezés (változó) elé zárójelben megadjuk, mire szeretnénk fordítás közben a szemlélt változót alakítani. Vagyis a szintaxis:

```
(típus) kifejezés
```

…ahol `típus` a típus neve, amire szeretnénk a kifejezést konvertálni. Például explicit konverziót kell végrehajtanunk a következő példában, ha `f`-be nem csak az egész számú osztás hányadosát szeretnénk betenni:

```
int a = 12, b = 5;
float f;
f = (float) a / (float) b;    // az f változó értéke 2.4 lesz
```

Ha kihagytuk volna a *cast*-okat, tehát csak azt írtuk volna, hogy `f = a / b;`, akkor az eredmény `2.00` lett volna, hiszen az osztás művelet mindkét operandusa egész szám (`int`), ezért az eredmény is az lett volna, hiába `float` a balérték.

Mint látjuk, a típuskonverziók gyakran lehetnek hibák forrása. Ezért óvatosan kell velük bánni, és ha csak lehet, mellőzni kell őket.

Adatbevitel és adatkiírás

Mint az algoritmusoknál már láttuk, praktikusan nem lehet elképzelni egy olyan programot, amiben nem szerepel legalább egy adatkiírás (a képernyőre) vagy adatbevitel (a billentyűzetről). Azonban a C nyelvben egyáltalán nincsen erre vonatkozó utasítás. Ez persze nem azt jelenti, hogy a C-ben nem lehetséges adatokat megjeleníteni a képernyőn, vagy bevinni valamit a billentyűzetről. A C mérnökei ugyanis ezt a hiányt úgy helyettesítették, hogy függvényeket hoztak létre, amik pont ezt a két feladatot teljesítik, és betették őket a C beépített függvénykönyvtárába. Az adatkiírás függvénye a `printf()`, az adatbevitel függvénye pedig a `scanf()`. Amint már az előfeldolgozónál említettük, ezeket a függvényeket viszont be kell olvasni a programba az `#include` direktívával. Ez a két függvény a `stdio.h` (ang. ***ST**andar**D** **I**nput **O**utput*, jelentése „standard bemenetel kimenetel”) nevezetű fájlban található. Tehát e két függvény használatához a program elejére a következő sort kell tenni:

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

A továbbiakban ezt a két függvényt fogjuk részletesebben bemutatni.

Adatkiírás a printf() függvénnyel

Mint tudjuk, a függvényhívás úgy történik, hogy a függvény neve után zárójelbe beletesszük a paramétereket, vesszővel elválasztva. A `printf()` függvényt **kétféleképp** lehet használni: **egyszerű sztring** (azaz szöveg) kiírására, valamint **sztring kiírására változókkal és konstansokkal**.

Először az **egyszerűbb verziót** fogjuk megnézni (egyszerű sztring kiírása). Ennek szintaxisa a következő:

```
printf(formátum);
```

…ahol `formátum` maga a szöveg **idézőjelbe** (`"`) téve, amit ki szeretnénk írni a képernyőre. Például:

```
printf("Hello World!");
```

Ez a következő üzenetet írja ki a monitorra:

```
Hello World!
```

Itt még annyit lehetne mondani, hogy a sztring bizonyos speciális jeleket, ún. **escape szekvenciákat** is tartalmazhat. Ezeket akkor használjuk, amikor *fehér* karaktereket (pl. új sor, tabulátor, stb.), vagy egyéb „foglalt” karaktereket szeretnénk a képernyőn megjeleníteni, mint pl. az idézőjelet (`"`), aposztrófot (`'`), kérdőjelet (`?`), stb. Az escape szekvenciák mindig **fordított osztásjellel** (ang. ***Backslash***, \) kezdődnek, majd valamilyen karakter követi őket. A fontosabbakat megtalálhatjuk a táblázatban.

Escape szekvencia	Jelentés
\n	új sor
\t	tabulátor
\'	aposztróf (')
\"	idézőjel (")
****	backslash (\)
\?	kérdőjel (?)

Nézzünk meg erre egy példát:

```
printf("Aposztrofot is irhatunk: \'.\nLatszik\?");
```

A képernyőre a következő üzenet lesz kiírva (két sorba a `\n` miatt):

```
Aposztrofot is irhatunk: '.
Latszik?
```

A `printf()` **bonyolultabb verziójával** már nem csak egyszerű szöveget, hanem konstansokat és a változók értékeit is kiírhatjuk a képernyőre. Szintaxisa:

`printf(formátum, argumentumlista);`

…ahol `formátum` a kiírandó szöveg **idézőjelbe** (**"**) téve, `argumentumlista` pedig a plusz kiírandó változók és konstansok, **vesszővel** (**,**) elválasztva. A listának tetszőleges számú tagja lehet. Talán legegyszerűbb lenne, ha rögtön egy konkrét (teljes értékű) példával kezdenénk:

```
#include <stdio.h>
main()
{
    int sum;
    sum = 50 + 18;
    printf("50 es 18 osszege %d\n", sum);
}
```

A programban létrehoztunk egy egész típusú változót (`sum`), majd egy összeget rendeltünk hozzá. A `printf()` formátumában most először találkozunk egy furcsa jellel: ez a **százalék** (**%**) és az utána írt **d** karakter. Ezzel egy ún. **speciális konverziót** adtunk meg, ami mindig **%** jellel kezdődik, majd bizonyos számú karakter követi (mi esetünkben egy). Ebben az esetben a `d` karakter **decimális** (tizes számrendszerben levő) számra utal, ami stimmel, hiszen a `sum` változó `int` típusú.

Ahogy a szintaxisból látszódik is, a formátum után jön az argumentumok listája. Ez ebben az esetben egy tagot tartalmaz, ez a `sum` változó. Amikor a programot lefuttatjuk, a következőt fogjuk látni a képernyőn:

```
50 es 18 osszege 68
```

A `printf()` függvényt lényegében a következőképp kell használni: először a formátumba (**idézőjelekkel** határolva) beírjuk azt az üzenetet, amit ki szeretnénk írni a képernyőre. Azokra a helyeken, amelyekre valamilyen változó (pl. `sum`) értékét szeretnénk megjeleníteni, speciális konverziót szúrunk be, amit mindig `%` jellel indítunk, majd valamilyen segédkarakterrel folytatunk (pl. `d`, ha a kiírandó változó típusa egész szám). A formátumba több változó is kerülhet, ekkor minden helyre kitesszük a speciális konverziókat. Viszont mi ezzel csak azt mondtuk el a C-nek, hogy ezekre a helyekre nem sima szöveget, hanem konkrét változók értékeit szeretnénk kiíratni, viszont a C nem tudja, melyik változókról van szó (a `%d`-ről csak az derül ki, hogy egész típusú változóról lenne szó, az viszont már nem, hogy konkrétan melyik változóról). Ezért a formátum után **vesszőt** kell tenni, és egymás után felsorolni azokat a változókat, amiket ki szeretnénk íratni. Viszont két dologra figyeljünk nagyon oda:

- A formátumba beszúrt speciális konverziók száma meg kell, hogy egyezzen az argumentumlista tagjainak számával.
- A sorrendre vigyázni kell. Tehát, amilyen sorrendben adtuk meg a formátumban a speciális konverziókat, olyan sorrendben kell a hozzájuk kapcsolódó változókat is megadni.

Nézzük meg például a következő programot:

```
#include <stdio.h>
main()
{
    int x, y, z;
    x = 10;
    y = 25;
    z = x * y;
    printf("A ket szam szorzata: %d * %d = %d", x, y, z);
}
```

Látjuk, hogy a fenti `printf()` formátumában most már három speciális (`%` jellel kezdődő) konverzió van, mindegyik `%d`. Viszont azt is lehet látni, hogy a formátum után az argumentumlista szintén három változót tartalmaz (`x`, `y` és `z`). A C az első `%d` helyére kiírja az argumentumlista első tagjának, azaz az `x` változónak az értékét, a második `%d` helyére az `y` értékét, míg a harmadik `%d` helyére a `z` változó értékét. Így a program futtatásakor a következő üzenet fog minket fogadni:

```
A ket szam szorzata: 10 * 25 = 250
```

Vigyázzunk: ha nem jó sorrendben adjuk meg a változókat az argumentumlistában, furcsa kimenetet kaphatunk. Így például ha a fenti `printf()`-ben felcserélnénk az `y` és `z` változókat az argumentumlistában, tehát azt írnánk, hogy:

```
printf("A ket szam szorzata: %d * %d = %d", x, z, y);
```

…akkor a következőt kapnánk a program futtatásakor:

```
A ket szam szorzata: 10 * 250 = 25
```

Persze ettől függetlenül a változók értéke jó (így `y` értéke továbbra is 25, míg `z` értéke 250), de az adatkiírás során elrontottuk a sorrendet.

Még annyit emelnénk ki, hogy az argumentumlista tagjai nem csak változók lehetnek, hanem konstansok, vagy akár olyan *kifejezések*, amik kiértékelésének eredménye konstans. Ez annyit jelent, hogy például olyan kifejezést is megadhatunk tagként, amiben számolni kell. pl. `x * y`. Ez egy kifejezés, amit először ki kell értékelni, vagyis ki kell számolni ennek eredményét, és a kapott eredményt (konstans formájában) a számítógép kiírja a képernyőre. Sőt, a kifejezés akár függvény is lehet, amennyiben a függvénynek megfelelő visszatérési értéke van. A függvényekről majd csak később lesz szó, de demonstrációként nézzük meg az előző példának egy olyan megoldását, aminél az argumentum utolsó tagja nem sima változó, hanem kifejezés, amit ki kell értékelni:

```
#include <stdio.h>
main()
{
    int x, y;
    x = 10;
    y = 25;
    printf("A ket szam szorzata: %d * %d = %d", x, y, x * y);
}
```

Mint látjuk, a `printf()` argumentumlistájának utolsó tagja most nem a `z` (nem is definiáltunk változót ilyen néven), hanem a „képzeletbeli” `z` kiszámolása magában a `printf()`-ben történik (`x * y`). Az eredmény ugyanaz lesz, mint korábban, azonban a háttérben megfigyelhetünk egy lényeges különbséget a két program működése között: míg az első programban létrehoztunk egy `z` változót, aminek értéke számolás után `250` lett, addig a második programban nem hoztuk létre ezt a változót, hanem „menet közben”, a `printf()` belsejében számoltuk ki az értéket. Azonban míg az első programban a `z` értéke (ami `250`) megmarad a program hátralévő részéig (a számítógép memóriájában), addig a második programban a `250` érték egy konstans, amit nem mentettünk el egy változóba se, így a számítógép memóriájába se, tehát az eredmény elveszik a `printf()` végrehajtása után. A `250` szám továbbra is ott lesz a képernyőn, de a komputer memóriájában biztos nem. Ez sokszor nem is gond, például ebben a példában sem, hiszen a kiírás után a program egyébként is véget ér, de ha olyan programot írunk, ahol a `250` értékre később is szükség lenne, akkor azt be kell másolni egy változóba (ezt tettük az első programban, ahol a `z`-be írtuk bele a számot). Ez persze nem kötelező, de ha a program futása során később is szükség lenne erre a számra, azt kénytelenek lennénk újból kiszámolni (`x * y`), ami fölöslegesen terhelné csak a processzort. A példatárban lesz olyan feladat, ami kimondottan ezt a szituációt mutatja majd be.

Természetesen a `%d` mellett még sok más speciális konverzió létezik, hiszen a változók nem csak egész típusúak lehetnek. Nekünk a legfontosabbak a következők lesznek:

- **`%d`** – tízes számrendszerű, tehát decimális, egész típusú (`int`) számok;
- **`%f`** – lebegőpontos (`float` és `double`) számok;
- **`%c`** – karakterek (`char`).

A speciális konverziókról egy kicsit bővebben (haladó szint)

A speciális konverziókkal nagyon szimpatikus dolgokat lehet véghezvinni, viszont használatuk bonyolult. Oldalakat kellene írni róluk, ha teljes egészében be szeretnénk őket mutatni. Mi megelégszünk egy rövidebb „kurzussal”, aminek sokan még hasznát vehetik.

Az első, amit le kell tisztázni, hogy jóval több konverzió létezik a korábban megemlített háromnál. Egy teljesebb lista a következő táblázatban látható:

Speciális konverzió	Az argumentum típusa	Jelentés
`%d`	`int`	tízes számrendszerű, azaz decimális előjeles egész szám
`%u`	`unsigned int`	előjel nélküli decimális egész szám
`%f`	`double`	lebegőpontos számok (float és double) kiírására
`%c`	`char`	karakter
`%s`	`char*`	sztring, azaz karakter tömb (szöveg)
`%%`	`nincs`	Nincs konverzió, csak kiírja a % jelet, ezért sokan escape szekvenciának tartják

Két megjegyzést tennénk ezzel a táblázattal kapcsolatban. Először, a `%f` konverziót nem csak `double`, hanem `float` típusú változók kiírására is használjuk. Másodszor, a `%s` konverziót **karaktertömb** (ún. **sztring**, ang. *String*) kiírására használjuk. A sztringek lényegében karaktersorozatok, úgy képzeljük el őket, mint sok `char` típusú változót egymás mellett. Ennek típusa `char*`.[18]

Speciális konverzióval nem csak különböző típusú változókat írhatunk ki a képernyőre, hanem azt is meghatározhatjuk, milyen „sablon” alapján nyomtassuk ki őket. Ez lényegében a formázást jelenti. Az előbb említett `%.2f` is egy ilyen formázás volt, ami azt határozta meg, hány tizedes pontossággal írja ki a lebegőpontos számokat a képernyőre. De ezen kívül még számos más formázási lehetőség létezik. Így például a C lefoglalhat bizonyos számú helyet a képernyőn a változóra, azt jobbra, vagy balra igazíthatja, kibővítheti az egész számokat nullákkal (pl. `25` helyett `000025`), stb. Mi ezeket nem fogjuk külön részletezni, viszont elmagyarázzuk, mit is jelent lényegében a `%.2f`. A szintaxist így lehetne leírni:

```
%<.pontosság>f
```

…ahol `.pontosság` egy pozitív egész szám (előtte egy **pont** jellel). `<>` jelekkel van körülvéve, ami annyit jelent, hogy opcionális (nem kötelező) részről van szó, tehát azt is írhatjuk, hogy `%f`, de ha szeretnénk a pontosságot is megadni, akkor ki kell tenni a pont jelet is. Tehát, ha egy lebegőpontos számot két tizedes pontossággal szeretnénk kiírni, akkor a következő speciális konverziót írjuk: `%.2f`. Egyébként, ha továbbra is a standard `%f` jelölést használjuk, akkor a C magától dönt a tizedesjegyek számáról, ami alapértelmezetten hat. Vigyázzunk, a lebegőpontos számoknál az egész számok esetében is kiíródnak a tizedes jegyek. Így pl. a `2` szám lebegőpontos kiírása `2.000000` (`%f` konverzióval). Ha ezt el akarjuk nyomni, akkor pontosságként írjunk nullát (`%.0f`).

18 A * szimbólum mutató típusra utal, erre külön nem térünk ki.

Adatbevitel a scanf() függvénnyel

Adatok bevitelére a `scanf()` függvény szolgál. Mint észre fogjuk venni, eléggé hasonlít a `printf()`-re, de persze vannak különbségek is. Szintaxisa a következő:

`scanf(formátum, argumentumlista);`

…ahol `formátum` **idézőjelbe** (**"**) téve meghatározza a beolvasás formátumát, és a beolvasott értékeket hozzárendeli az `argumentumlista` tagjaiként megadott változókhoz. Míg a `printf()`-nél arra kértük a számítógépet, hogy írja ki a képernyőre az idézőjelbe tett szöveget, addig a `scanf()`-nél a számítógép kéri a felhasználót, hogy írja be az idézőjelbe tett szöveget a billentyűzeten keresztül. Kezdjük megint egy konkrét példával:

```
#include <stdio.h>
int main()
{
    int a;
    char c;
    printf("Irj be egy egesz szamot es egy betut:");
    scanf("%d,%c",&a,&c);
    printf("A kovetkezoket irtad be: %d, %c", a, c);
}
```

Látjuk, hogy a formátum a `scanf()`-nél is tartalmaz speciális konverziókat. A `%d` megint egész számra utal (most viszont beolvasás céljából), a `%c` pedig egy karakterre. Azt is látjuk, hogy a két konverzió **vesszővel** (`,`) van elválasztva. Ez nem kötelező, de fontos megjegyezni, hogy ahhoz, hogy a `scanf()` függvény végrehajtása sikeres legyen, a formátumban minden olyan karaktert, ami nem a konverzió része, vagy nem helyköz, kötelezően be kell írni a billentyűzeten keresztül. Mivel a vessző nem a konverzió része, ezért a beolvasás a fenti példában akkor lesz sikeres, ha a két változó értékét beírás közben vesszővel választjuk szét, például beírjuk, hogy `15,K`, majd lenyomjuk az *enter*-t. Ezzel lényegében arra köteleztük a felhasználót, hogy a két beolvasni kívánt változó értékét vesszővel választja szét egymástól. Ha a felhasználó nem így tesz, rossz lesz a beolvasás.

Ha ez nem tetszik, a konverziókat elválaszthatjuk más jellel (vagy jelekkel). Gyakori például a **helyközzel** ( ) való elválasztás. Ezt kétféleképp írhatjuk:

```
scanf("%d %c",&a,&c);
```

…vagy…

```
scanf("%d%c",&a,&c);
```

Mint látjuk, az első esetnél helyközzel választottuk szét a két speciális konverziót, míg a másodiknál semmivel (egybe írtuk a konverziókat). Lényegében mindegy, melyik írásmódot választjuk, de sokan mégiscsak a helyközzel való szétválasztást ajánlják, mert egyrészt olvashatóbb, másrészt jobban kihangsúlyozza, hogy helyközzel kell a változók értékeit szétválasztani. Ahhoz, hogy a beolvasás sikeres legyen, most tehát úgy írnánk be például a két változó értékét, hogy `15 K`.

A `scanf()`-nél használatos speciális konverziók praktikusan megegyeznek a `printf()` konverzióival, így azokat nem vennénk át megint, viszont ki kell térnünk egy nagyon fontos

kivételre. A `printf()`-nél úgy említettük, hogy a `%f` konverziót használjuk `float` és `double` típusú változók kiírására is. A `scanf()`-nél viszont ez nem érvényes, mivel a **`%f`** konverzió csakis és kizárólag **`float`** típusú változók beolvasására szolgál. Ha **`double`** típusú változót szeretnénk beolvasni, akkor a **`%lf`** konverziót használjuk (az angol *Long Float*, azaz „hosszú *float*” rövidítésből ered, ami lényegében megegyezik a `double` jelentésével).

Ami az argumentlistát illeti, itt lényegében ugyanaz ismétlődik, mint a `printf()` függvénynél, tehát minden egyes konverziót a formátumból a C hozzárendel egy változóhoz a listából, és persze ügyelnünk kell arra, hogy a konverziók száma a formátumból megegyezzen az argumentumlista tagjainak számával, valamint a sorrend is fontos. Azonban egy fontos különbséget is meg kell említeni: a `scanf()` definíciója szerint az argumentumlista nem a változók azonosítóját várja, hanem azok **címét** a memóriában. Ez annyit jelent, hogy minden változó (tehát argumentum) elé **`&`** jelet kell tenni.[19] Ez alól viszont kivétel a sztring beolvasása, mivel az már eleve mutató-típusú, így annál nem kell `&` jelet írni.

19 A mutató (Pointer) operátoroknál említettük, hogy nem fogunk velük bőven foglalkozni, mégis itt szükség van rájuk.

Az alapvető könyvtári függvények

Mint láttuk, a C önmagába véve nem sok plusz lehetőséggel rendelkezik, viszont a készítők rengeteg függvényt írtak hozzá, amiket bármikor meghívhatjuk. Kettővel már megismerkedtünk, ezek a `printf()` és `scanf()` függvények a **`stdio.h`** könyvtárból. Most meg fogunk ismerkedni pár másik függvénnyel, amik inkább a matematikai számolásban lehetnek segítségünkre. Egyébként minden könyvtárfüggvény hasonlóságuk szerint van csoportosítva a könyvtárakba. Minket ebben a pillanatban két ilyen könyvtár érdekel: `stdlib.h` és `math.h`.

A **`stdlib`** könyvtár függvényeit használjuk az abszolút érték kiszámolására, véletlen számok generálására, memóriahasználatra, különböző konverziókra, stb. Minket csak az első kettő fog érdekelni. A leggyakrabban használt függvények a `stdlib` könyvtárból a következők:

`stdlib` könyvtár			
Függvény neve	**Paraméter(ek) típusa**	**Az eredmény típusa**	**Jelentés**
`abs(x)`	`int`	`int`	Az x szám abszolút értékét adja eredményül, tehát |x|
`rand()`	–	`int`	Véletlenszám-generátor
`srand(n)`	`unsigned int`	–	A rand() véletlenszám-generátort inicializálja

A másik fontos matematikai függvényeket tartalmazó könyvtár a **`math`**. Trigonometriai függvényeket, exponenciális és logaritmusos függvényeket, hatványozást, gyökvonást, abszolút értéket és még sok mást tartalmaz. A legfontosabbak a következők:

`math` könyvtár			
Függvény neve	**Paraméter(ek) típusa**	**Az eredmény típusa**	**Jelentés**
`cos(x)`	`double`	`double`	Az x szám koszinuszát adja eredményül
`sin(x)`	`double`	`double`	Az x szám szinuszát adja eredményül
`tan(x)`	`double`	`double`	Az x szám tangensét adja eredményül
`exp(x)`	`double`	`double`	Az e^x kifejezés értékét adja eredményül
`log(x)`	`double`	`double`	Az x természetes alapú logaritmusát adja eredményül. Vigyázzunk: x > 0
`log10(x)`	`double`	`double`	Az x tízes alapú logaritmusát adja eredményül. Vigyázzunk: x > 0
`pow(x, y)`	`double, double`	`double`	Az x szám y hatványkitevőre emelt értékét adja eredményül, tehát x^y
`sqrt(x)`	`double`	`double`	Az x négyzetgyökét adja eredményül
`ceil(x)`	`double`	`double`	A legkisebb olyan egész számmal tér vissza, amelyik nem kisebb x-nél
`floor(x)`	`double`	`double`	A legnagyobb olyan egész számmal tér vissza, amely nem nagyobb x-nél
`fabs(x)`	`double`	`double`	Ugyanaz, mint az abs(x) a stdlib könyvtárból, tehát az x abszolút értékét adja eredményül, csak ez nem egész, hanem lebegőpontos számokkal dolgozik (a float abs rövidítése)

A leggyakrabban használt függvények az **`abs()`** a `stdlib` könyvtárból, a `math` könyvtárból pedig a **`pow()`**, **`sqrt()`** és a **`fabs()`**.

Szekvenciális és elágazó struktúrájú programok

A fejezet előző részeiben átvettük a C programozási nyelv alapvető részeit: az előfeldolgozót, a kommentárokat, változókat, konstansokat, operátorokat és az adatbevitel, valamint adatkiírás függvényeit. Ez a tudás már elég ahhoz, hogy elkezdjük a „legegyszerűbb" C programok írását, és ez alatt a szekvenciális, azaz lineáris struktúrájú programokat értjük. Ennél a struktúránál nem lesz elméleti rész, hiszen jól tudjuk az *Algoritmusok* fejezetből, mi is a szekvenciális struktúra.

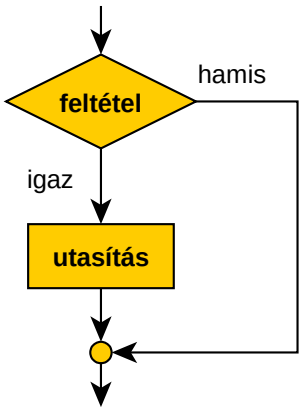
Azonban már az algoritmusok gyakorlása közben rájöhettünk, hogy egyszerű szekvenciális struktúrákkal nem sokra mehetünk. Mint tudjuk, az elágazások, vagy szelekciók arra szolgálnak, hogy változtassunk a feladat végrehajtásán, attól függően, hogy milyen feltételek teljesültek. Lényegében ez annak felel meg, amikor egy úton haladva elágazáshoz jutunk, és fel kell mérnünk, azaz döntenünk kell, hogy melyik úton haladunk tovább, természetesen azzal a feltétellel, hogy vissza nem fordulhatunk. Az algoritmusoknál a döntéshozatalt **döntéscsomóponttal** oldottuk meg, amibe bele kellett írni azt a *feltételt*, azt a *kérdést*, aminek eredménye eldönti, merre haladunk tovább. Mint tudjuk, a feltételre, azaz kérdésre két választ tudtunk adni: **igaz**, vagy **hamis** választ.

A C-ben két mechanizmus létezik elágazások létrehozására: az egyik az `if`, a másik pedig a `switch`.

Az if utasítás

Az **`if`** utasítással (ang. *if*, jelentése *„ha"*) lényegében **egy-** vagy **kétágú elágazásokat** tudunk létrehozni. Először természetesen az **egyágú** verziót tekintjük meg. Ennek szintaxisa a következő:

```
if (feltétel)
    utasítás
```



…ahol `feltétel` egy kifejezés, ahova kell beírni a kérdést, `utasítás` pedig egy utasítás.[20] Például:

```
if (x > 0)
    printf("Ez egy pozitiv szam");
printf("\nVege.");
```

20 Fel lehet tenni a kérdést, miért nincs az utasítás után pontosvessző, miközben tudjuk, hogy az kötelező (a példában is ki van téve). Emlékezzünk vissza az *Operátorok* alfejezet elejére: ott megemlítettük, hogy minden utasítás legtöbbször kifejezésből áll, amit pontosvessző követ. Tehát, ha szigorúan a szintaxist nézzük, akkor az `utasítás` ugyanaz, mintha azt írtuk volna, hogy `kifejezés;` . Ez majd főleg a feltételes operátor tárgyalásánál lesz fontos.

Tehát, mint látjuk, a program, mielőtt eldöntené, melyik ágon halad tovább, először leellenőrzi az `x` értékét. Ha `x` nagyobb nullánál, akkor a feltétel értéke igaz, vagyis az *„igaz”* ágon haladunk tovább, tehát kiírjuk az „`Ez egy pozitiv szam`” üzenetet a képernyőre. Ha viszont `x` értéke `0`, vagy negatív szám, akkor egyszerűen átugorjuk az előbb említett utasítást (lényegében a *„hamis”* ágon haladunk végig, de most az nem tartalmaz egy utasítást sem, tehát üres). Mindkét esetben viszont kiíródik a „`Vége`” üzenet, hiszen az már az ágak *újbóli* találkozása után helyezkedik el, tehát nem az elágazás része.

Azonban sokszor megesik, hogy egy elágazásba nem csak egy, hanem *több* utasítást is szeretnénk tenni, viszont a C csak egy utasítást fogad el. Ebben az esetben az utasítást ún. **összetett utasításra**, vagyis **utasításblokkra** kell cserélni, ami lehetővé teszi, hogy annyi utasítást tegyünk egy elágazásba, amennyit akarunk. A blokkot **kapcsos zárójelek**, azaz **`{`** és **`}`** jelek közé kell tenni. Felhívnánk a figyelmet, hogy míg az egyszerű utasítás végére mindig pontosvesszőt kell tenni, addig a blokk után ez teljesen felesleges. Nézzünk meg most erre egy példát:

```
if (x > 0)
{
    printf("Ez egy pozitiv szam");
    x += 10;
}
printf("\nVege.");
```

Mint látjuk, a fenti programrészlet „igaz” ága most blokkra lett cserélve, ami két utasításból áll (üzenet kiírása, majd `x` értékének növelése tízzel). Ha a feltétel hamis, akkor az egész blokk át lesz ugorva (hamis ág még most is hiányzik). Mindkét esetben a „`Vége`” üzenet kiírásával folytatódik a program futása, mivel az már nem az elágazás része.

Mi már lényegében találkoztunk kapcsos zárójelekkel, hiszen minden programunkban az `int main()` után ki kellett nyitni, a program végén pedig be kellett csukni a kapcsos zárójelet. Ez is egy blokk, méghozzá a program blokkja. Erre majd később, a függvények fejezetben térünk ki.

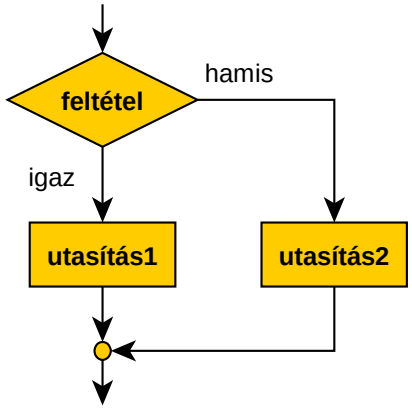
Két dolgot kell még kiemelni:

- A könnyű áttekinthetőség érdekében érdemes a blokk utasításait *beljebb húzni*. A legtöbb *IDE* ezt automatikusan elvégzi.
- Sokan még az egy utasításból álló ágat is blokkba teszik, mivel később, ha a programot bővíteni kell, a kapcsos zárójeleket így is, úgy is ki kellene tenni, mellesleg kisebb lesz a hibalehetőség is.

Az if-else szerkezet

Természetesen **kétágú** szelekciót is létrehozhatunk. C-ben a hamis ág jelölésének kulcsszava az **`else`** (ang. *else*, jelentése *„különben”*). Ennek szintaxisa a következő:

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



Tehát, ha a feltétel értéke *„igaz”*, akkor `utasítás1` fog végrehajtódni, különben pedig `utasítás2` (ha a feltétel értéke *„hamis”*). Természetesen itt is blokkba kell tenni az ágakat, ha több utasítást szeretnénk végrehajtani egy ágon belül. A következő példában a számítógép kiírja a megfelelő üzenetet, attól függően, hogy mennyi az `x` értéke, valamint vagy hozzáad az értékéhez tízet (ha pozitív szám), vagy elvesz (különben). Azt is vegyük észre, hogy a **„Vege”** üzenet mindenféleképp ki lesz írva, hiszen az `if-else` struktúra után helyezkedik el:

```
if (x > 0)
{
    printf("Ez egy pozitiv szam");
    x += 10;
}
else
{
    printf("Ez vagy 0, vagy negativ szam");
    x -= 10;
}
printf("\nVege.");
```

A feltételes operátor (? :) (haladó szint)

A programok írása során sokszor találkozhatunk rövid elágazásokkal, vagyis olyan `if` szerkezetekkel, amiknek *„igaz”* és *„hamis”* ága is csupán egyetlen utasításból áll. Mivel ezek a szerkezetek igen gyakoriak, a C-be bevezettek egy operátort azzal a céllal, hogy relatív egyszerűen és röviden lehessen az efféle `if` szerkezeteket írni. Ez az *Operátorok* részben már említett **feltételes operátor**, aminek jelölése **`? :`** .

Mint már tudjuk, a feltételes operátor az egyedüli *háromoperandusú* operátor C-ben. Szintaxisa:

A feltételes operátor szintaxisa	Az if-else szerkezet szintaxisa

`feltétel ? kifejezés1 : kifejezés2;`	`if (feltétel)`
	`  utasítás1`
	`else`
	`  utasítás2`

Tehát, ha `feltétel` értéke igaz, akkor `kifejezés1` fog végrehajtódni (az `if-else` szerkezet szintaxisánál ezt `utasítás1`-ként jelöltük), ha pedig a válasz hamis, akkor pedig `kifejezés2`.[21] Mint látjuk, míg a hagyományos `if-else` szerkezet több sort foglal el, addig a feltételes operátorral csupán egyetlen sorba kifértünk, a kód pedig továbbra is olvasható.

Három dologra illik odafigyelni feltételes operátor használata esetén:

- Csak olyan `if` szerkezetet helyettesíthetünk feltételes operátorral, aminek **mindkét ága** (*„igaz”* és *„hamis”*) létezik![22]
- Nem elég, hogy az eredeti `if` szerkezet mindkét ága létezzen, az is kötelező, hogy mindkét ága **egy utasításból** álljon!
- Ha nem vagyunk biztosak a feltételes operátor elsőbbségében és asszociativitásában, tegyük úgy a feltételt, mint a két kifejezést is (`kifejezés1` és `kifejezés2`) **zárójelbe**!

Szemléltessük a feltételes operátor használatát egy példán keresztül (mint látjuk, a feltétel és mindkét kifejezés zárójelben van):

Hagyományos if-else szerkezettel	Feltételes operátorral

`if (x > 0) {`
`  x = x + 10;`
`}`
`else`
`{`
`  x = x - 10;`
`}`
	`(x > 0) ? (x = x + 10) : (x = x – 10);`

Habár még elég sok dolgot lehet a feltételes operátorral véghezvinni, mi ezen a szinten meg fogunk elégedni ennyivel. Most pedig térjünk vissza az `if-else` szerkezethez egy kicsit komolyabb szinten.

21 A feltétes operátor szintaxisánál nem írhattuk volna `kifejezés1` helyett azt, hogy `utasítás1`, mivel tudjuk, hogy az utasítások végére pontosvesszőt kell tenni, itt viszont azt **tilos** (csak a sor végén van ott a pontosvessző, mivel a feltételes operátor is egy nagyobb utasítás).

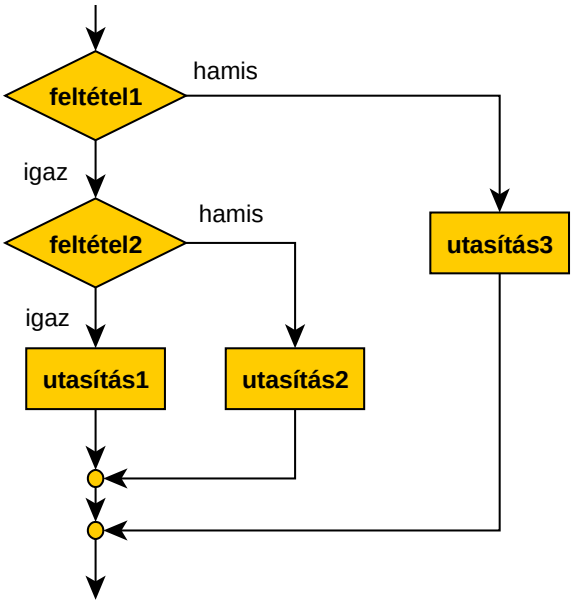
22 Habár a C későbbi verzióiba beletették, hogy nem kötelező mindkét ág kifejezését kitölteni (meghatározott kikötésekkel), mi erre nem fogunk kitérni, helyette igyekezzünk az operátort az eredeti szabályok szerint használni!

If utasítások egymásba ágyaztatása, az else-if szerkezet, és az összetett feltételek írása (haladó szint)

Eddig minden elágazás blokkja szekvenciális (azaz lineáris) szerkezetű volt, azonban fel lehet tenni a kérdést, hogy vajon ezek az ágak (blokkok) lehetnek-e újból elágazók. A válasz igen. Tehát, ha akarjuk, az elágazás ágaiba újabb elágazásokat tehetünk, és azok mellékágaiba újabb elágazásokat. Új elágazások kerülhetnek az *„igaz”* és *„hamis”* ágba is. Azonban a félreértelmezések és hibák elkerülése érdekében ajánlatos ebben az esetben mindig használni a blokkok behatárolására szolgáló kapcsos zárójeleket. Az egymásba ágyaztatott `if` utasítások szintaxisa intuitív, és tetszőlegesen alakíthatjuk kedvünk szerint. A két legegyszerűbb szituáció az, amikor vagy az *„igaz”* (tehát `if`) ágba teszünk egy új ágat, vagy a *„hamis”* (tehát `else`) ágba:

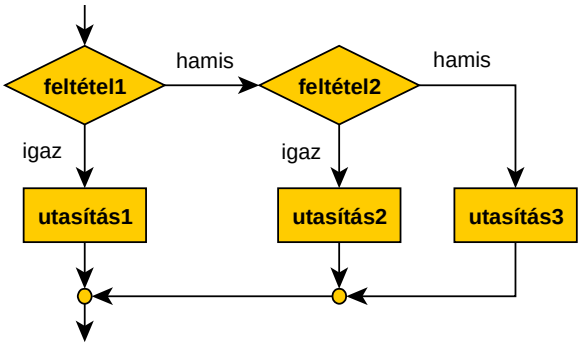
If („igaz”) ágba való ágyaztatás

```
if (feltétel1)
{
    if (feltétel2)
    {
        utasítás1
    }
    else
    {
        utasítás2
    }
}
else
{
    utasítás3
}
```



Else („hamis”) ágba való ágyaztatás

```
if (feltétel1)
{
    utasítás1
}
else
{
    if (feltétel2)
    {
        utasítás2
    }
    else
    {
        utasítás3
    }
}
```



Sokan észre fogják venni, hogy az `else` ágba való ágyaztatás úgy néz ki, mintha *háromágú* elágazást hoztunk volna létre, pedig lényegében arról van szó, hogy az `else` ágba beágyaztunk egy újabb elágazást. Ha ennek az `else` ágába újabb `if` struktúrát szúrtunk volna be, akkor már *négyágú* elágazást kaptunk volna. Mivel ezek a struktúrák elég gyakoriak, ezért a C-ben egy könnyítést vezettek be, hogy ne kelljen minden új ágat beágyazással megoldani. Ez az **`else-if`** szerkezet, ami lényegében azt teszi lehetővé, hogy egy szintre, egyetlen nagyobb `if` struktúrával több ágat hozzunk létre, ágyaztatás nélkül. Ennek szintaxisa három ág esetén a következő:

```
if (feltétel1)
{
    utasítás1
}
else if (feltétel2)
{
    utasítás2
else
{
    utasítás3
}
```

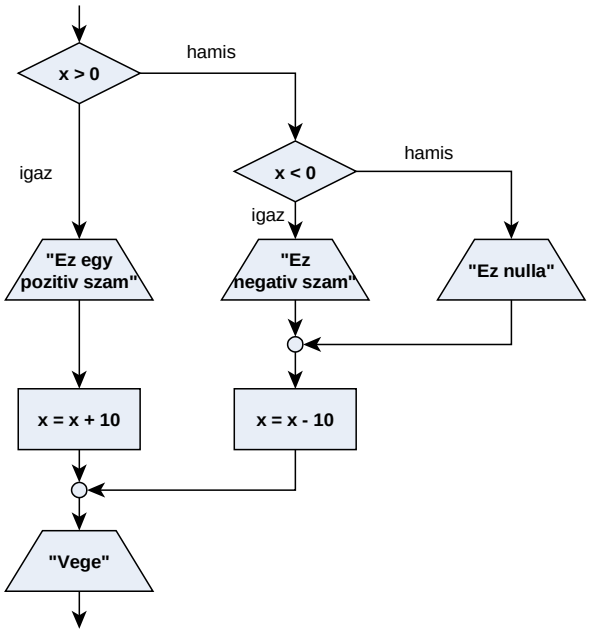
Ne feledjük:

- Az utolsó `else` ág léte nem kötelező;
- A struktúrában annyi `else if` ágat használhatunk, amennyit akarunk;
- A struktúrában legfeljebb csak egy `else` ág lehet.

Nézzünk meg egy példát kétféleképp megoldva, először `else` ágba való ágyaztatással, majd `else-if` szerkezettel:

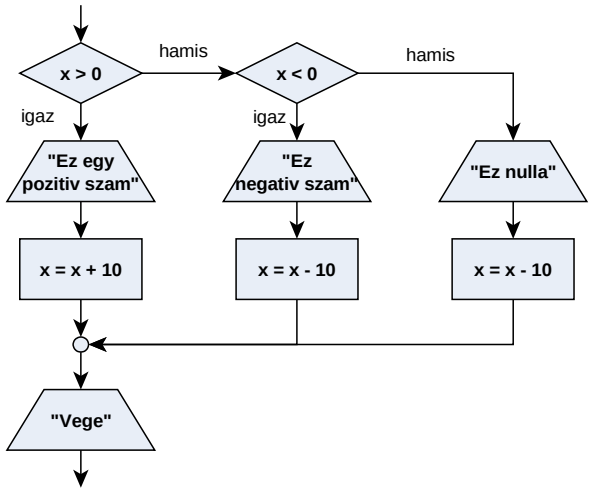
Else ágba való ágyaztatással

```
if (x > 0) {
    printf("Ez egy pozitiv szam");
    x += 10;
}
else {
    if (x < 0) {
        printf("Ez negativ szam");
    }
    else {
        printf("Ez nulla");
    }
    x -= 10;
}
printf("\nVege.");
```



Else-if szerkezettel

```
if (x > 0) {
    printf("Ez egy pozitiv szam");
    x += 10;
}
else if (x < 0) {
    printf("Ez negativ szam");
    x -= 10;
}
else {
    printf("Ez nulla");
    x -= 10;
}
printf("\nVege.");
```



Vegyük észre, hogy az `else` ágba való ágyazáskor (első oszlop) az `x -= 10;` utasítás nem része a belső `if`-nek, mivel mindkét esetben (ha `x` kisebb 0-nál, vagy ha egyenlő 0-val) le kell futtatni. Természetesen be is tehettünk volna a belső `if` mindkét ágába, de ezzel csak utasításduplikációt értünk volna el. Azonban az `else-if` szerkezetnél (második oszlop) nincs választásunk. Mivel az `else-if` szerkezet igazi ágakat hoz létre, ezért ott kénytelenek vagyunk külön beírni mindkét ágba (`x` kisebb 0-nál és `x` egyenlő 0-val) az `x -= 10;` utasítást. Ez nagyon jól látszódik a mellékelt folyamatábrákon is.

A fenti példát nézve még azt említenénk meg, hogy kompaktság miatt sok programozó a blokk kezdetét jelző `{` jelet nem teszi új sorba, hanem az előző sor végére egy helyköz után.

Mielőtt továbblépnénk, még meg kellene említeni, hogy a feltételek írásakor lehetőségünk van **összetett feltételt** is írni. Ugyanis a matematikával ellentétben egy C-kifejezésbe csak egy

darab összehasonlító operátor (`<`, `>`, `<=`, `>=`, `==` és `!=`) kerülhet. Ha többre van szükség, akkor két lehetőségünk van:

- Beágyaztatunk egy új `if` struktúrát a fő `if` *„igaz”* ágába;
- Összetett feltételt írunk, úgy, hogy a kifejezéseket **logikai operátorokkal** (`&&`, `||`) kötjük össze.

Ennek szemléltetésére a legjobb példa az, amikor azt kell kitanulmányozni, hogy x beleesik-e az $[a, b]$ intervallumba, vagy sem. Matematikában ez könnyen megoldható, hiszen ott csak leírjuk, hogy $a < x < b$. Azonban, ez nem engedélyezett C-ben. Helyette két kisebb kifejezésre kell bontani, és azokat összekötni valamilyen logikai operátorral (ebben az esetben logikai *ÉS*-sel). Ezt a matematikában úgy írnánk, hogy $a < x \wedge x < b$, C-ben pedig úgy, hogy `a < x && x < b`. Vagyis:

If ágba való ágyaztatással

```
if (a < x) {
    if (x < b) {
        printf("x eleme [a, b]-nek");
    }
else {
    printf("x nem eleme [a, b]-nek");
}
```

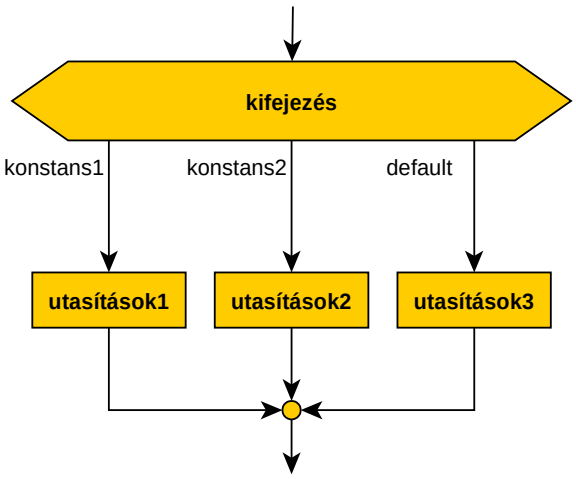
Összetett feltétellel

```
if (a < x && x < b) {
    printf("x eleme [a, b]-nek");
}
else {
    printf("x nem eleme [a, b]-nek");
}
```

A switch utasítás (haladó szint)

A **`switch`** utasítás **többirányú** programelágazásokat tesz lehetővé, persze pár kikötéssel. Szintaxisa a következő:

```
switch (kifejezés)
{
    case konstans1:
        utasítások1
        break;
    case konstans2:
        utasítások2
        break;
    default:
        utasítások3
        break;
}
```



Amint azt láthatjuk a folyamatábrán, először kiértékelődik a `kifejezés` értéke, majd a C fentről lefelé haladva megkeresi azt az esetet (`case` ágat), aminek konstansa megegyezik `kifejezés` értékével. Tehát, ha `kifejezés` értéke egyenlő `konstans1`-gyel, akkor azt a `case` ágat kezdi végrehajtani, különben átugorja, és megnézi, hogy akkor egyenlő-e `konstans2`-vel, stb. Ha

`kifejezés` értéke egyik `case` ág konstansával sem egyezik, akkor az alapértelmezett `default` ág fog végrehajtódni. Azonban pár dologra fel kell hívni a figyelmet:

- A szintaxisban található `kifejezés` csak egész típusú szám (`int`), vagy karakter típus (`char`) lehet, mivel a karakterek is lényegében egész számokra vezethetők vissza.
- Egy `switch` szerkezetben annyi `case` ág lehet, amennyire csak szükségünk van (igazából ez implementációfüggő).
- Az alapértelmezett `default` ág léte nem kötelező. Ebben az esetben, ha `kifejezés` értéke nem egyenlő egy `case` ág konstansával sem, akkor egyik ág sem lesz végrehajtva. Ez ugyanaz, amikor egy `if` szerkezetnek nincs `else` ága, viszont a feltétel értéke hamis lett.
- A szintaxisnál észrevehettük, hogy minden `case` ág az utasítások után egy `break;` utasítással ér véget. Erre azért van szükség, mert a `switch` lényegében a `goto` vezérlésátadó utasítást maszkírozza. A `break` lehetővé teszi, hogy a kiválasztott `case` ág utasításainak végrehajtása után a program a `switch` szerkezet mögött folytatódjon (a gyűjtőcsomópont után a folyamatábrán). Így érjük el azt, hogy a `switch` szerkezetből legfeljebb csak egy ágat hajtsunk végre.

A következő példa az `x` egész típusú változó értékétől függően különböző aktivitásokat fog végrehajtani. Három águnk van: arra az esetre, ha `x` értéke `1`, ha `x` értéke `0` és az összes többi értékre (`default` ág).

```
switch (x) {
    case 1:
        printf("x erteke 1");
        res = res * 10;
        break;
    case 0:
        printf("x erteke 0");
        res = 0;
        break;
    default:
        printf("x erteke nem 1, se nem 0");
        res = res * 50;
        break;
}
```

Kicsit bővebben a break utasításról (haladó szint)

Habár a legtöbb esetben a `switch` utasítást úgy használják, ahogy azt nemrég elmagyaráztuk, de érdemes megemlíteni, hogy a `break` utasítás léte nem kötelező. Ha nem lenne `break` a `case` ág végén, akkor a C az ág végrehajtása után rögtön a következő ágba „esne”, és ott folytatná a program futtatását (ráadásul tovább nem is hasonlítaná össze a kifejezés értékét a másik `case` ág konstansával). Mivel legtöbbször nem ez a célunk, ezért lett már a szintaxisba bevezetve a `break` utasítás. Persze vannak esetek, amikor ajánlott több ág „összefolyatása”, például amikor egy kifejezés több értékére is szeretnénk ugyanazt az ágat végrehajtani. Mivel a `switch` szerkezetnél nem lehet összetett feltételeket írni (pl. a `&&` logikai operátorral), ezért lényegében más lehetőségünk nincs is. Ekkor szorosan egymás alá tesszük ezeket az ágakat, és csak az utolsó `case`

ágnál fogjuk megtölteni az ágat utasításokkal, a többi ágat teljesen „üresen” hagyjuk. Például tételezzük fel, hogy szeretnénk kiírni egy meghatározott szöveget, amikor a felhasználó a **„i”** betűt írja be, de nem tudjuk, hogy a felhasználó kis- vagy nagybetűt fog beírni:

```
switch (betu) {
    case "i":
    case "I":
        printf("A valasz IGEN.");
        break;
    default:
        printf("A valasz NEM.");
        break;
}
```

Ciklusos struktúrájú programok (iterációk)

Az algoritmusoknál tanult programstruktúrák közül a harmadikat a **ciklusok** képviselték (a szekvenciális és elágazós struktúrák után). A ciklus lényegében egy olyan utasítás (vagy utasítások sorozata), amit többször is végrehajthatunk. Sok problémát nagyon nehéz, vagy szinte lehetetlen lenne ciklusok nélkül megoldani. Azonban a ciklusok sok hiba forrását is képezik. Ezért minden ciklushoz egy **döntéscsomópontot** is tettünk, amiben valamilyen **feltételt** számoltunk ki, és aminek igaz, vagy hamis értéke határozta meg, hogy beléphetünk-e a ciklusba, vagy sem. Ha a döntéscsomópont nem létezne, akkor a ciklusból soha nem léphetnénk ki, vagyis a programunk futása soha nem fejeződne be. Ezt **végtelen ciklusnak** nevezzük, és minden programozó feladata, hogy a programja soha ne lépjen végtelen ciklusba. Ezért minden ciklus két kötelező részből áll:

- **feltételből** és
- **ciklustestből**.

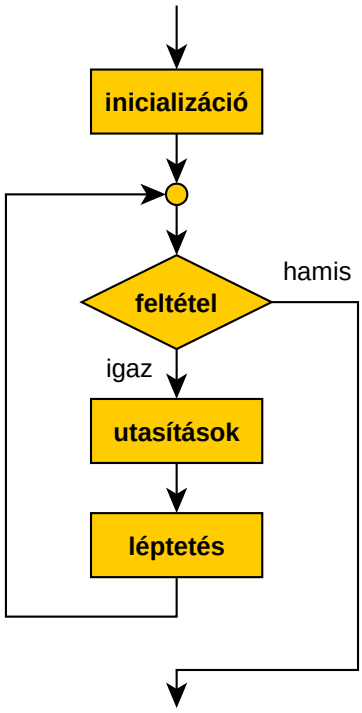
Természetesen a feltétel léte még nem garancia arra, hogy nem fog végtelen ciklus létrejönni, ezért a feltételt és a ciklustestet mindig körültekintően kell írni.

A C-ben háromféle utasítás létezik ciklusok realizációjára: `for`, `while` és a `do-while`. Ezek közül az első kettő *elöltesztelős* (a feltétel a ciklustest előtt található), míg az utolsó *hátultesztelős* (a feltételt a ciklustest után írjuk). A továbbiakban mindhárom utasítást részletesen be fogjuk mutatni.

A for utasítás

A `for` egy **előltesztelős** ciklusutasítás, és a leggyakrabban akkor használjuk, amikor előre tudjuk, hogy a ciklustestet hányszor akarjuk végrehajtani. Az utasítás szintaxisa a következő:

```
for (inicializálás; feltétel; léptetés)
{
    utasítások
}
```



Mint látjuk, a `for` utasítás zárójelében három kifejezés található, amiket **pontosvesszővel** (`;`) választunk el egymástól:

- **`inicializálás`** – az algoritmusos feladatoknál láttuk, hogy a feltétel megírásához szükséges bevezetni valamilyen segédváltozót, ún. **ciklusváltozót** (de nevezik még **léptető-változónak** is), aminek értékétől fog függni a feltétel logikai értéke. Ezt a segédváltozót sokszor **`i`**-nek, vagy **`j`**-nek neveztük. Az `inicializálás` kifejezés feladata az, hogy ezt a ciklusváltozót inicializálja, azaz valamilyen kezdőértéket adjon neki. A fenti folyamatábrán jól látható, hogy ez a kifejezés csak egyszer fut, amikor először jutunk a ciklus elé.
- **`feltétel`** – az algoritmusoknál ismert döntéscsomópontot ebbe a kifejezésbe kell írni. Ez lényegében a **kérdés**, amit felteszünk magunknak mindig, mielőtt belépnénk a ciklustestbe. Ha a feltétel értéke igaz, akkor beléphetünk a ciklustestbe, különben kiesünk a ciklusból. A feltétel írásánál valamilyen összehasonlító operátort is szokás használni (`<`, `>`, `<=`, `>=`, `==` és `!=`).
- **`léptetés`** – ha visszaemlékezünk az algoritmusos feladatokhoz, láttuk, hogy a ciklustest utolsó utasításaként a ciklusváltozót egy új értékre kellett tenni, aminek új, friss értéke lesz felhasználva az új feltétel kiértékeléséhez. Ez legtöbb esetben a ciklusváltozó eggyel való növelése, vagy csökkentése volt. A `for` ciklusnál nem kell ezt az utasítást a ciklustestbe tenni, hiszen megvan neki a saját helye: ez a `léptetés` kifejezés.

Habár látszódik a szintaxisból, érdemes megemlíteni, hogy a `for` kulcsszó után írt zárójel bezárása után (a szintaxis első sora) **nem teszünk pontosvesszőt**.

A `for` ciklus, mint ahogy látható a fenti algoritmusnál is, a következőképp működik: amikor a ciklusutasításhoz érünk, legelőször kiértékelődik a ciklusváltozó értéke az `inicializálás` kifejezésben, majd jön a `feltétel` kifejezés kiértékelése, ahol megnézzük, hogy a kérdés értéke igaz, vagy hamis. Ha **igaz**, beléphetünk **egyszer** a ciklustestbe. A ciklustest utasításainak végrehajtása után a `léptetés` kifejezés értékelődik ki, azaz itt történik a ciklusváltozó új értékének kiszámolása, és ezzel lényegében befejeztünk egy ún. **iterációt (ciklusmenetet)**. Ahhoz, hogy másodszorra is belépjünk a ciklusba, megint kiértékelődik a `feltétel` kifejezés, és ha a válasz ismét igaz, beléphetünk másodszor is a ciklustestbe. A ciklusból akkor vágódunk ki, amikor egyszer a `feltétel` kifejezés értéke **hamis** lesz.

Példaként írjunk egy rövid programot arra vonatkozóan, hogy a számítógép annyiszor írja ki a „`Hello world`” üzenetet, amennyiszer a felhasználó megadta a program elején:

```
#include <stdio.h>
int main()
{
    int x, i;
    printf("Hanyszor irjam ki a hello uzenetet?");
    scanf("%d", &x);
    for (i = 1; i <= x; i++)
    {
        printf("Hello world!\n");
    }
    return 0;
}
```

Még pár érdekesség a for ciklusról (haladó szint)

Egy haladóbb funkciója a `for` ciklusnak az, hogy ha akarunk, tehetünk belé több `inicializálás`, több `feltétel` és több `léptetés` kifejezést is. Egy kitűnő példája ennek az, amikor több ciklusváltozót kell használnunk egyszerre (pl. `i` és `j`). A sorrend továbbra is fontos: először az `inicializálás` kifejezéseket kell megadni, majd a `feltétel` kifejezéseit, végül a `léptetés` kifejezéseit. Az egyforma típusú kifejezéseket azonban **vesszővel** (`,`) kell szétválasztani. Az alábbi példában a ciklusnak két ciklusváltozója van (`n` és `i`) és ez miatt két `inicializáció` és `léptetés` kifejezése, de csak egy `feltétel` kifejezése (ami használja mindkét változót):

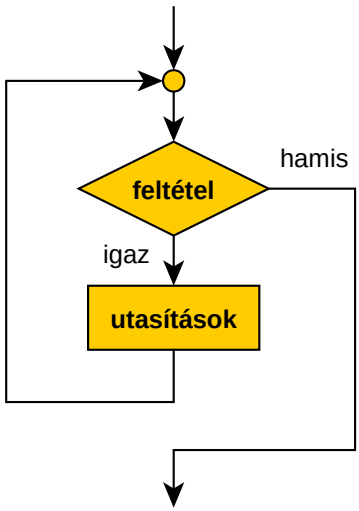
```
for (n = 0, i = 100; n != i; ++n, --i) {
    // ciklustest
}
```

A `for` ciklus testébe olyan utasításokat teszünk, amiket csak szeretnénk. Az `if` utasításnál említett ágyaztatás itt is működik. Tehát, ha az a célunk, bírunk a `for` belsejébe újabb `for`-t tenni, de akár `if` elágazásokat is.

A while utasítás

A **`while`** szintén egy **előltesztelős** ciklusutasítás, azonban a `for` utasítással ellentétben nem kötelezi a ciklusváltozó létét. A `while` utasítást általában akkor használjuk, amikor nem tudjuk előre meghatározni azt, hányszor fogunk a cikluson végigmenni. A szintaxis a következő:

```
while (feltétel)
{
    utasítások
}
```



A `for` utasítással összehasonlítva most már látható, hogy a `while` sokkal szabadabb, nem köti meg annyira a programozó kezét, hiszen nincs benne se inicializálás, se léptetés. Lényegében csak a két kötelező rész található meg benne: a feltétel és maga a ciklustest. Habár látszik a szintaxisból, érdemes megemlíteni, hogy a feltétel zárójelének bezárása után **nem szabad pontosvesszőt tenni**.

A `while` utasítás nem szorul bő magyarázatra. A ciklus addig ismétlődik, amíg a `feltétel` kifejezés értéke **igaz**. Tehát amikor először a ciklus elé kerülünk, kiértékelődik a `feltétel` értéke. Ha az érték igaz, beléphetünk a ciklustestbe. A ciklustest utasításainak végrehajtása után megint a feltételhez kerülünk, aminek értéke másodszor is kiértékelődik. Ha a válasz ismét igaz, akkor

másodszor is beléphetünk a ciklustestbe. A folyamat addig folytatódik, míg egyszer a feltétel kifejezés eredménye **hamis** lesz. Ekkor kiesünk a ciklusból.

Vegyük észre, hogy megtörténhet olyan eset, hogy **egyszer sem** fogjuk a ciklustestet végrehajtani! Ez akkor történik meg, amikor a `feltétel` értéke a ciklushoz való érkezéskor rögtön hamis.

Habár a `while` sokkal szabadabb használatot engedélyez a `for` utasításnál, a hibák (és végtelen ciklusok) lehetősége is nagyobb, hiszen magunknak kell gondoskodni a ciklusváltozó inicializálásáról és léptetéséről. Szerencsére csak annyi plusz feladatunk van, hogy a szükséges parancsokat be kell szúrni, persze a megfelelő helyre. Tehát, ha `while` utasítást használunk ciklusváltozóval, akkor a következőkre kell odafigyelni:

- **Inicializálás** – a ciklusváltozót a ciklustestben való használat előtt inicializálni kell, azaz a helyes kezdőértékre kell tenni. Ezt a parancsot nem szabad a ciklustestbe helyezni, mert akkor minden ciklustest végrehajtásakor felülíródna az értéke (pl. mindig `0`-ra változna az értéke, hiába a léptetés). Ez végtelen ciklust eredményezne. Helyette a ciklus **elé** kell tenni az inicializálást (nézzük meg a `for` utasítás folyamatábráját, látni fogjuk, hogy az inicializálás lényegében ott is a ciklustest előtt helyezkedik el).
- **Léptetés** – a ciklusváltozó értékét a ciklustesten való végighaladás után változtatni kell (azaz léptetni). Ez a `for`-nál mindig a ciklustest után, de a feltétel előtt történt. A `while` utasítás nem tartalmaz külön részt a léptetésre, ezért itt ki kell bővíteni a ciklustestet ezzel a lépéssel. A parancsot legjobb a ciklustest végébe szúrni (ez legyen a ciklustest utolsó utasítása).

Illusztrációképp alakítsuk át a `for` ciklusutasításnál említett példát (**„Hello world”** üzenetek kiírása) úgy, hogy most `for` helyett `while` ciklusutasítást használjon:

```
#include <stdio.h>
int main()
{
    int x, i;
    printf("Hanyszor irjam ki a hello uzenetet?");
    scanf("%d", &x);
    i = 1;    /* az i ciklusváltozó inicializálása while előtt  */
    while (i <= x)
    {
        printf("Hello world!\n");
        i++;  /* a ciklusváltozó léptetése a ciklustest végén   */
    }
    return 0;
}
```

Mint látjuk, csak annyi volt a feladatunk, hogy a megfelelő helyre kellett szúrni a ciklusváltozó inicializálását és léptetését. Ne feledjük, minden `for` utasítást átalakíthatunk `while`-ra, és fordítva. Hogy melyiket használjuk, az a programozó döntése.

Vannak azonban szituációk, amikor a `while` használata okosabb megoldás, mint a `for`, mint például akkor, amikor fölösleges a ciklusváltozó jelenléte. Ennek tipikus példája az, amikor a felhasználót kötelezni szeretnénk, hogy egy meghatározott billentyűt üssön be. (például a felhasználót ne engedjük tovább, míg le nem nyomja az **„E”** billentyűt). Mivel a felhasználó

kiismerhetetlen, ezért itt nincs nagy értelme ciklusváltozót használni. Helyette feltételnek azt tesszük be, hogy a felhasználó milyen billentyűt nyomott le.

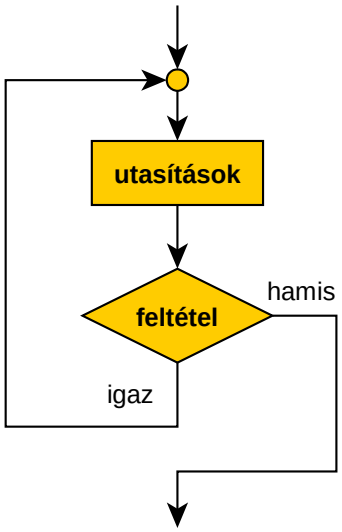
Ha már az egyik ciklusutasítás másikba való átalakításáról van szó, fel lehet tenni a kérdést, hogyan alakítsunk át `while` utasítást `for` utasításra, ha nem használunk ciklusváltozót. Ez a kérdés már azért is érdekes, mert a `for` kifejezetten erőlteti a ciklusváltozó használatát. Habár a részletekre nem térünk ki, legalább két megoldás létezik: az egyik az, hogy bevezetünk egy színlelt (üres) ciklusváltozót, amit a megszokott módon léptetünk (pl. mindig növeljük), azonban a feltételben nem használjuk. Másik megoldás az, hogy nem is használunk a `for`-ban ciklusváltozót, azaz a `for` utasítás zárójelében kihagyjuk az `inicializálás` és `léptetés` kifejezéseket, és csak a `feltétel` kifejezést hagyjuk meg. Ez engedélyezett, azonban a C-nek tudnia kell, hogy melyik rész melyik kifejezésnek felel meg, ezért továbbra is ki kell tenni a pontosvesszőket a zárójel belsejében. Például:

```
for ( ; betu == 'E'; ) {
    // ciklustest
}
```

A do-while utasítás

A **`do-while`** utasítás nagyon hasonlít a `while`-ra, egy lényeges különbséggel: a feltétel nem a ciklustest előtt, hanem **mögött** van, tehát ez egy **hátultesztelős** ciklusutasítás. Azonban minden más megegyezik. A szintaxisa:

```
do
{
    utasítások
}
while (feltétel);
```



Mint látjuk, a legnagyobb különbség a `while` és `do-while` között az, hogy a `feltétel` kifejezés a ciklustest mögé került. Ennek az a következménye, hogy a ciklustestet **legalább egyszer** biztosan végig fogjuk hajtani! Ez logikus is, hiszen amikor először érkezünk a szerkezethez, a ciklustesten egyszer végighaladunk, és csak aztán jutunk a feltételhez. Ha a `feltétel` értéke igaz, akkor még egyszer visszamehetünk és végigmehetünk a ciklustesten (most már másodszor), ami után megint a `feltétel` elé kerülünk. Azonban ha egyszer a feltételre hamis választ adunk, kirepülünk a ciklusból.

Habár a szakirodalom `do-while` néven emlegeti ezt a ciklusutasítást, az utasításpárost (a `do` és `while` utasításokat) soha nem írjuk egybe, és nem kötőjellel választjuk el őket egymástól (lásd a szintaxist). A szerkezet a `do` kulcsszóval indul, ami után jön a ciklustest. A ciklustest bezárása után jön a `while` kulcsszó, majd a `feltétel`, amit zárójelbe teszünk. A `while` ciklusutasítással ellentétben itt a feltétel zárójelének bezárása után már kötelezően ki kell tenni a pontosvesszőt.

A `while` utasításnál már említettük, hogy a `for` utasítás könnyedén alakítható `while`-ra, és fordítva. Ugyanez vonatkozik a `do-while` utasításra is. Tehát, a három ciklusutasítás lényegében ugyanaz, csak más megközelítésből próbálják a problémát megoldani. A teljesség kedvéért alakítsuk át az előző **„Hello world”** példát úgy, hogy most a `do-while` utasítást használja:

```
#include <stdio.h>
int main()
{
    int x, i;
    printf("Hanyszor irjam ki a hello uzenetet?");
    scanf("%d", &x);
    i = 1;    /* az i ciklusváltozó inicializálása do-while előtt  */
    do
    {
        printf("Hello world!\n");
        i++;  /* a ciklusváltozó léptetése a ciklustest végén      */
    }
    while (i <= x);  /* hátultesztelős feltétel */
    return 0;
}
```

Mint látjuk, könnyű a dolgunk: a ciklusváltozó inicializálása és léptetése identikus, csak a feltétel most a ciklustest végére került, a ciklustestet pedig a `do` kulcsszóval kezdtük. Gondot csupán az okozhat, ha a feltétel értéke rögtön a ciklushoz való érkezéskor hamis. Ekkor a `while` egyszer sem fogja a ciklustestet végrehajtani, míg a `do-while` egyszer mindenféleképpen. Ezt kiküszöbölhetjük egy egyszerű `if` elágazással a ciklustest elején.

Habár a statisztika azt sugallja, hogy a `do-while` utasítást relatív ritkán használják az emberek, mégis vannak szituációk, amikor pont ez adja a legelegánsabb megoldást. Akkor használatos, amikor szeretnénk a ciklust legalább egyszer végrehajtani, vagy egyéb okunk van rá, hogy a feltételt a ciklustest mögé rejtsük.

Tömbök

Tételezzük fel, hogy egy olyan feladatot kell megoldani, amiben a felhasználónak be kell írnia egy osztály tanulóinak évlezáró osztályzatait informatikából, ami alapján a számítógép kiírja a képernyőre egyenként a diákok osztályzatait, majd az egész osztály átlagát ebből a tantárgyból. A megoldás látszólag egyszerűnek tűnik: hozzunk létre annyi változót, amennyi diák van az osztályban „egy diák – egy változó” elven. Itt azonban komoly problémába ütközünk, ugyanis attól függően, hogy mekkora az osztály létszáma, akár több tíz változót kellene létrehoznunk. Habár ezt lehetséges megoldani, mi lenne, ha az egész iskola tanulóit kellene számolni? A probléma forrása az, hogy az eddigiek során tanult változók csak egyetlen érték tárolását teszik lehetővé. Tehát, ha definiálunk egy tetszőleges típusú változót, annak az idő egy pillanatában csak egy értéke lehet.

Ahhoz, hogy ezt a problémát megoldjuk, szükség lenne arra, hogy a hasonló objektumokat **csoportosítsuk**. Lényegében egy olyan változót szeretnénk létrehozni, aminek nem egy, hanem több értéke lehetne. A mi esetünkben például létrehozhatnánk az előbb említett osztály csoportját, amibe beletehetnénk az osztály minden diákjának osztályzatát. Azonban az osztályzatok beírásakor fontos a diákok napló szerinti **sorszáma**. Ezért ez a csoport nem lehet valamiféle halmaz, hiszen ott az elemek sorrendje nem játszik szerepet. Helyette a csoportnak inkább egy **sorozatnak** kellene, hogy legyen, ahol minden diákot elérhetünk a napló szerinti sorszáma segítségével. Ezt a sorozatot akár vizuálisan is lerajzolhatjuk (tételezzük fel, hogy az osztálynak tíz tanulója van):

diák

5	**3**	**2**	**5**	**4**	**4**	**3**	**2**	**4**	**2**
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*

A sorozat neve *diák*, a téglalapok alján található kis dőlt számok egytől tízig a diákok *sorszáma* a naplóban, míg a téglalapok belsejében a diákok *osztályzata* található (az *adat*). Az efféle elrendezés azért jó, mert bármely pillanatban hozzáférkőzhetünk bármely diákhoz (azaz annak osztályzatához) a naplóbeli sorszáma alapján. Így például látjuk, hogy a 7. sorszámú diák osztályzata 3.

A C-ben és a többi programnyelvben ezeket a struktúrákat **tömböknek (ang. *Array*)** nevezzük, de gyakori még a **vektor** kifejezés is. Lényegében annyi a különbség a fenti vizuális jelölés és a C-s jelölés között, hogy a tömb neve természetesen nem tartalmazhat ékezetes betűket, a másik (talán lényegesebb) különbség viszont az, hogy a C-ben az **elemek sorszámozása nem 1-től, hanem 0-tól kezdődik**. Vagyis az előző példa a következőre alakul át:

diak

5	**3**	**2**	**5**	**4**	**4**	**3**	**2**	**4**	**2**
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Tehát, mint látjuk, az *első* elem sorszáma 0 (nem 1), az *utolsó* elem sorszáma pedig 9 (nem 10). Ha általánosítanánk, azt mondanánk, hogy ha a tömb mérete (elemeinek száma) n, akkor a tömb első elemének sorszáma **0**, az utolsó elemének sorszáma pedig ***n*-1**. Egyébként a sorszámot a programnyelvekben **indexnek** nevezzük. Az előzőleg emlegetett 7. sorszámú elem indexe tehát 6.

Fontos megjegyezni, hogy az egyszerű változókhoz hasonlóan a tömbnek is van **típusa**. A tömb típusa meghatározza, hogy a tömb elemei egyenként milyen típusúak lesznek (`int`, `float`, `char`, stb.). Egy tömbön belül az elemek típusa **nem keverhető**, vagyis a tömb minden elemének

azonos típusúnak kell lennie. Tehát, ha például egy egész típusú (`int`) tömböt hozunk létre, akkor annak minden eleme `int` típusú lesz. Látjuk, hogy a fenti `diak` tömb is `int` típusú, hiszen az elemek egész számok.

A tömbök deklarációja

Hasonlóképp, mint az egyszerű változókat, a tömböket is *deklarálni* kell használat előtt. A deklarációhoz, azaz a létrehozáshoz a C-nek három dologra van szükség: a tömb *nevére* (azonosító), *típusára* és *méretére*, azaz hosszára. A méret ebben az esetben azt határozza meg, hogy hány elemből fog állni a tömb. A deklaráció szintaxisa igen egyszerű:

```
típus tömbnév[méret];
```

…ahol `típus` a tömb típusa, `tömbnév` a tömb azonosítója, azaz neve, `méret` pedig annak hossza. Mint látjuk, a tömb méretét **szögletes zárójelek** (tehát `[` és `]`) közé kell zárni. A tömbnév és a szögletes zárójel között nem lehet helyköz. A nemrég említett 10 elemből álló egész számokat tartalmazó `diak` tömböt tehát így lehetne deklarálni:

```
int diak[10];
```

A tömb elemeihez való hozzáférés

Természetesen a most létrehozott tömb üres, használatra még nem alkalmas. A következő feladat tehát a tömb elemekkel való feltöltése (ha vizuálisan nézzük, akkor ez lényegében a téglalapok belsejébe való írás). Ez azonban igen egyszerű, hiszen elég csak tudni a tömb *nevét* és a megfelelő elemsorszámot (*indexet*). A szintaxis a következő:

```
tömbnév[index] = érték;
```

Mint látjuk, a fenti *értékadás* nagyon hasonlít az egyszerű változók értékadására, csak most az index megadása is fontos (amit szintén **szögletes zárójelbe** kell tenni). Tehát, az egyenlőség jobb oldalán található *értéket* hozzárendeltük a tömb megadott *indexű* eleméhez. Fontos tudni, hogy ha a tömb adott indexű eleme már tartalmazott valamilyen más értéket, az felülíródik (pont ahogy az alaptípusoknál is). Töltsük fel most a `diak` tömböt adatokkal úgy, hogy az megegyezzen a fenti vizuális példával:

```
diak[0] = 5;
diak[1] = 3;
diak[2] = 2;
diak[3] = 5;
diak[4] = 4;
diak[5] = 4;
diak[6] = 3;
diak[7] = 2;
diak[8] = 4;
diak[9] = 2;
```

Természetesen a tömbök adatokkal való feltöltésének van egy egyszerűbb és sokkal rövidebb módja is, amit majd hamarosan megnézünk, de előtte nézzük még meg, hogyan lehet megtudni, azaz *kiolvasni* a már adatokkal feltöltött tömb bizonyos indexű elemét. Az alaptípusokhoz hasonlóan elég csak megemlíteni a tömb *nevét*, de most az *indexére* is szükség van. Például írassuk ki a képernyőre az első diák osztályzatát:

```
printf("Az elso diak osztalyzata: %d", diak[0]);
```

Innen látjuk, hogy a tömbök konkrét elemeit ugyanúgy használjuk, mint az egyszerű változókat, csak most az index megadása is szükséges.

Most pedig nézzük meg, hogyan lehet egy tömböt könnyedén feltölteni adatokkal.

A tömbök definiálása

Még korábban említettük, hogy a deklarálás nem ugyanaz, mint a definiálás. A definiálás a tömbök esetében tartalmazza nem csak a tömb létrehozását, hanem annak inicializálását, azaz kezdőértékekkel való feltöltését is. A C-ben ez a tömbök esetében nagyon egyszerű, egy sorba belefér. A szintaxis:

`típus tömbnév[méret] = {érték1, érték2, …};`

Az előző deklaráció és a mostani definíció között az a különbség, hogy most a tömb nevén, típusán és méretén kívül rögtön megadjuk annak elemeit is, méghozzá **kapcsos zárójelbe** (tehát **`{`** és **`}`**), az elemeket pedig **vesszővel** választjuk el egymástól. Így az előzőleg demonstrált igen hosszúkás feltöltés most tíz sor helyett belefér csupán egyetlen sorba:

```
int diak[10] = {5, 3, 2, 5, 4, 4, 3, 2, 4, 2};
```

Problémák elkerülése érdekében mindig ügyeljünk arra, hogy a tömb mérete, valamint a kezdőértékek száma megegyezzen. Ha *több* kezdőértéket írunk be, mint amekkora a tömb mérete, a C hibaüzenetet fog kiírni. Azonban ha *kevesebb* kezdőértéket adunk meg, mint amekkora a tömb mérete, akkor a C *sorban* feltölti a tömb elemeit a megadott kezdőértékekkel, a maradék kezdőérték nélküli elemet pedig vagy 0-ra teszi, vagy határozatlan marad. Ez sok hiba forrását okozhatja, ezért ehhez az utóbbihoz csak akkor folyamodjunk, ha tudjuk, mit csinálunk.

Amennyiben a tömböt definiáljuk, azaz rögtön *inicializáljuk* is, akkor kihagyhatjuk a tömb méretét (viszont a szögletes zárójeleket továbbra is **kötelezően ki kell írni**). Tehát a `diak` tömböt akár így is megadhatjuk:

```
int diak[] = {5, 3, 2, 5, 4, 4, 3, 2, 4, 2};
```

Mint látjuk, a szögletes zárójelbe most nem tettünk semmit. A C fordítás közben megszámolja, hány kezdőértéket írtunk be a kapcsos zárójelbe, és pont akkora tömböt fog létrehozni. Tehát lényegében így is egy 10-tagú tömb lesz létrehozva.

A tömbök és a ciklusok

A tömbök a valóságban hatalmas méretűek lehetnek, sokkal nagyobbak, mint a 10 elemből álló `diak` tömb. Tételezzük fel, hogy most ki szeretnénk a képernyőre írni a tömb összes elemének tartalmát. Ha csak a `diak` tömböt nézzük, akkor lényegében 10 darab `printf()` függvényre lenne szükségünk. Ez még kivitelezhető, de mit tegyünk, ha a tömb mérete több száz, vagy ezer? Itt jönnek előtérbe a ciklusok. Mint már tudjuk, a ciklusok egy bizonyos programkód körkörös végrehajtását teszik lehetővé, és ezért ideálisak a tömböknél. Tehát, a mi esetünkben tegyük a `printf()` függvényt a tömb belsejébe, majd hajtsuk végre a tömb belsejét annyiszor, amekkora a tömb mérete (vagyis annyiszor, amennyi elem van a tömbben). Hasonlóképp ciklust használhatunk arra is, amikor a felhasználónak kell a billentyűzeten keresztül beírni egyesével a tömb elemeit. Ekkor a `scanf()` függvényt tennénk a ciklus belsejébe.

A három ciklus közül a `for` ciklus használata ajánlott, hiszen a ciklusváltozó ideális az index megadására. Érdekességként megemlítenénk, hogy a ciklusváltozót a legtöbb esetben `i` betűvel azonosítottuk (még mielőtt megismerkedtünk volna a tömbökkel), de ez nem véletlen, hiszen lényegében az index kifejezés első betűje. A lenti példában a könnyebb érthetőség kedvéért még `index`-ként fogjuk a ciklusváltozót elnevezni, de minden más esetben már a rövidített `i` azonosítót fogjuk használni.

Például írjunk egy programot, ami arra kéri a felhasználót, hogy írja be a billentyűzeten keresztül egy 10-tagú tömb elemeit, majd írja ki a számítógép a tömb összes elemét a képernyőre (ezt sokszor *listázásnak* nevezzük). A program kinézete a következő:

```
#include <stdio.h>
#define MERET 10

int main()
{
    int tomb[MERET];
    int index;
    for(index = 0; index < MERET; index++) {
        printf("tomb[%d] = ", index);
        scanf("%d", &tomb[index]);
    }
    for(index = 0; index < MERET; index++) {
        printf("A tomb %d. eleme: %d\n", index, tomb[index]);
    }
    return 0;
}
```

A fenti programhoz még annyit fűznénk hozzá, hogy a program egy `MERET` nevű *konstanst* is definiál (a `#define` direktívával), ami lényegében a tömb méretét tartalmazza. Ez hasznos, mert akkor a programban minden olyan helyen, ahol meg kell adni a tömb méretét, elég csak a konstanst használni. Ha később megváltozna a probléma alapfelállása és pl. a tömb méretét tízről 35-re kellene növelni, akkor nem kellene a program belsejében minden helyen átírni a méretet, hanem elég lenne csak a program elején a `MERET 10` kifejezést átírni `MERET 35`-re. A fenti programban háromszor kellett a tömb méretét megadni (a tömb deklarálásánál, valamint a két `for` ciklus feltételének megadásánál), de egy komolyabb és hosszabb programban ez a szám sokkal nagyobb lenne.

Felhívnánk a figyelmet, hogy a tömbök esetében sok hiba forrását az képezi, hogy rosszul adtuk meg a tömb indexét. Legtöbbször arról van szó, hogy nem létező indexű elemhez akartunk

hozzáférni (pl. a 10-tagú `diak` tömb 10. indexű eleméhez, ami lehetetlen). Ezt az **indexhatár átlépésének** nevezzük. Ez azért hiba, mert a `diak` tömb utolsó elemének indexe 9, és nem 10. Sajnos a C nem nyújt valami nagy segítséget az ilyen hibák megtalálására, sokszor csak onnan vesszük észre, hogy rosszul működik a program.

Mint látjuk, a ciklusok rettenetesen megkönnyítik a tömbök használatát, mondhatni kölcsönös egyetértésben élnek. A tömbökkel rengeteg komoly problémát lehet megoldani, de szinte egyiket sem tudnánk elképzelni ciklusok használata nélkül. A tömbök és ciklusok együttes használata azért is jó, mert a tömböket könnyű vizuálisan elképzelni, és így hozzájárulnak a ciklusok mélyebb elsajátításához.

A többdimenziós tömbök (haladó szint)

Mint láthattuk, a tömbökkel egy igen hatékony eszközt kapunk különböző problémák megoldására, de vannak esetek, amikor még ez sem elég. A tömbök ismertetésénél említett példát előhozva, mit tehetnénk akkor, ha egy osztály diákjainál nem csak egy tantárgyból kellene az évlezáró osztályzatot elmenteni, hanem több tantárgyból? Létrehozhatnánk külön tömböt az összes tantárgyra, vagy használhatnánk egy ún. többdimenziós tömböt.

A többdimenziós tömb az egyszerű tömb általánosítása. Ugyanis tudjuk, hogy a tömb belső elemeinek típusa egyforma kell, hogy legyen, például egész számok (`int`), karakterek (`char`), valós számok (`float`), stb. Azonban a C-ben, mint sok más programnyelv esetében is, ennél tovább mentek, például a tömb elemei ne csak alaptípusok (egyszerű típusok, pl. `int`, `float`, stb.) lehessenek, hanem akár **tömbök** is. Így jutottak a többdimenziós tömbökig, amit legegyszerűbben így definiálhatunk: a **többdimenziós tömb** egy olyan tömb, aminek elemei szintén tömbök.

Attól függően, hogy hányszor vannak a tömbök egymásba ágyaztatva, annyi dimenziós tömböt kapunk. Az elnevezés onnan ered, hogy a többdimenziós tömb legegyszerűbb formája, amikor a tömb elemei olyan tömbök, amik elemei már alaptípusúak, vizuálisan úgy néz ki, mintha az egyszerű tömbhöz még egy dimenziót csatoltunk volna. Például:

0	**2**	**6**	**21**	**4**	**-5**
1	**8**	**15**	**32**	**-10**	**0**
2	**20**	**17**	**14**	**31**	**9**
3	**-9**	**-18**	**25**	**11**	**-27**
4	**-50**	**46**	**30**	**-14**	**52**
	0	*1*	*2*	*3*	*4*

A tömbön kívül található 0-tól 4-ig terjedő számok az *indexeknek* felelnek meg (mint mindig, **nullától** indulnak). Ezeket a tömböket **kétdimenziós tömböknek** nevezzük. Ha eltöröljük az egyszerű tömbök közti kis helyet és összenyomjuk az elemeket, a következő vizuális kinézetet kapjuk:

0	**2**	**6**	**21**	**4**	**-5**
1	**8**	**15**	**32**	**-10**	**0**
2	**20**	**17**	**14**	**31**	**9**
3	**-9**	**-18**	**25**	**11**	**-27**
4	**-50**	**46**	**30**	**-14**	**52**
	0	*1*	*2*	*3*	*4*

Mivel a fenti kép úgy néz ki, mintha egy mátrix lenne, ezért a kétdimenziós tömböket még **mátrixoknak** is nevezzük. Mint látjuk, a fenti kép egy 5x5 dimenziójú mátrixot mutat. Az első dimenzió ebben az esetben a sorokat jelképezi, a második dimenzió pedig az oszlopokat. Természetesen ez csak egy vizuális segítség, a sorokat és oszlopokat fel is cserélhetjük (tehát úgy, hogy az első dimenzió az oszlopokat jelképezze), de akkor tartjuk is magunkat ehhez. Felhívnánk a figyelmet, hogy a többdimenziós tömb dimenziói különbözőek is lehetnek. Az előző példában csak az egyszerűség kedvéért használtunk 5x5 dimenziójú tömböt, de a valóságban a dimenziók legtöbbször nem lesznek azonosak, pl. 4x6, 10x22, stb.

Azokat a tömböket, amik elemei tömbök, amik elemi szintén tömbök, amik elemei már alaptípusúak, **háromdimenziós tömböknek** nevezzük.

Az egyszerűség kedvéért az eddig tanult egyszerű vagy sima tömböket, vagyis azokat a tömböket, amik elemei alaptípusúak, **egydimenziós tömböknek**, vagy **vektoroknak** nevezzük. Ha összehasonlítjuk ezek vizuális kinézetét a kétdimenzióssal, valóban úgy néznek ki, mintha csak *egy dimenziójuk* lenne.

A C-ben lényegében nincsen szabály arra vonatkozóan, hogy hány dimenziós tömböt hozhatunk létre, azonban a leggyakoribbak az egy- és kétdimenziós tömbök, a háromdimenziós tömbök használata már jóval ritkább. Az ennél több dimenziójú tömböknek lényegében nincs gyakorlati használatuk, ezért sokszor csak ***n*-dimenziós tömböknek** nevezzük őket, ahol *n* a dimenziók száma.

A továbbiakban mi a kétdimenziós tömbökkel fogunk kicsit bővebben foglalkozni, de természetesen az itt említett dolgok kiterjeszthetők a három-, vagy *n*-dimenziós tömbökre is.

A kétdimenziós tömbök **deklarálása** nagyon hasonlít az egydimenziós tömbök deklarálásához. A különbség annyi, hogy a második dimenzió méretét is meg kell adni. Tehát a szintaxis:

```
típus tömbnév[méret1][méret2];
```

…ahol `típus` a kétdimenziós tömb típusa, `tömbnév` a tömb neve, `méret1` az egyik dimenzió mérete, `méret2` pedig a másik dimenzió mérete. Ezt lényegében úgy képzelhetjük el, mint egy `méret1` nagyságú tömböt, amik elemei `méret2` nagyságú tömbök. A fenti vizuális képen, ami egy 5x5 dimenziójú mátrixot mutat, a két dimenzió értéke ugyanaz (ezeket matematikában *négyzetes mátrixoknak* nevezzük), de ahogy már említettük, a két méret különbözhet is. Példák:

```
int matrix[5][5];
float diak[10][3];
```

A fenti példában a `diak[10][3]` egy olyan tömb, aminek mérete 10, és ennek elemei egyenként három-elemű tömbök. Ez a tömb ideális lehet az előzőleg feltett probléma megoldására. Ez a tömb

10 diák évlezáró osztályzatait tartalmazhatná 3 tantárgyból. Ha több tantárgyra lenne szükség, csak meg kellene növelni a második dimenzió értékét.

Mielőtt rátérnénk arra, hogyan lehet egy kétdimenziós tömb elemeihez hozzáférni, nézzük meg, hogyan lehet egy ilyen tömböt **definiálni**, azaz létrehozni és rögtön inicializálni is. Tételezzük fel, hogy a következő 3x5 dimenziójú mátrixot szeretnénk létrehozni:

$$\begin{bmatrix} 2 & 6 & 21 & 4 & -5 \\ 8 & 15 & 32 & -10 & 0 \\ 20 & 17 & 14 & 31 & 9 \end{bmatrix}$$

Ezt kétféleképp tehetjük meg:

- **Sorfolytonosan** – az elemeket egymás után, vesszővel elválasztva adjuk meg:

```
int tomb[3][5] = {2, 6, 21, 4, -5, 8, 15, 32, -10, 0, 20, 17, 14, 31, 9};
```

  Azonban a könnyebb áttekinthetőség miatt érdemes új sorba tenni az új sorok elemeit, tehát így:

```
int tomb[3][5] = { 2,  6, 21,   4, -5,
                   8, 15, 32, -10,  0,
                  20, 17, 14,  31,  9};
```

- **Sorokat kihangsúlyozva** – minden „altömböt” (azaz sort) külön kapcsos zárójelbe teszünk, amiket vesszővel választunk el egymástól:

```
int tomb[3][5] = {{2, 6, 21, 4, -5}, {8, 15, 32, -10, 0}, {20, 17, 14, 31, 9}};
```

  Természetesen a sorokat itt is lehet új sorba írni, tehát:

```
int tomb[3][5] = {{ 2,  6, 21,   4, -5},
                  { 8, 15, 32, -10,  0},
                  {20, 17, 14,  31,  9}};
```

A többdimenziós tömbök esetében nem ajánlatos a méret megadása nélküli inicializálás, mint az egydimenziós tömbök esetében, és a legtöbb esetben nem is engedélyezett. Ez logikus is, hiszen ha például a sorfolytonos inicializálásnál nem adnánk meg a tömb méretet, nem tudnánk, mekkora annak egyik és másik dimenziója!

Ami a kétdimenziós tömb **elemeihez való hozzáférést** illeti, az elemekbe való írás szinte ugyanaz, mint az egydimenziós tömböknél is, csak most mindkét dimenzió indexét meg kell adni. Tehát a szintaxis:

`tömbnév[index_i][index_j] = érték;`

…ahol `tömbnév` a tömb neve, `index_i` az első dimenzió (sor) indexe, `index_j` pedig a második dimenzió (oszlop) indexe. Így például az előző példa `tomb[1][2]` elemének értéke 32, vagyis a tömb egyes indexű sorának második indexű eleme. Az indexek minden dimenzióban 0-tól indulnak.

Fel kellene hívni a figyelmet, hogy ha egy kétdimenziós tömbnél csak az első indexet (a sort) adjuk meg (például `tomb[1]`), akkor lényegében a kétdimenziós tömb alapelemét kapjuk meg, ami ebben az esetben nem alaptípus, hanem tömb (vizuálisan nézve a mátrix egyik sora). Az

előző példát felhozva, a `tomb[1]` értéke szintén egy tömb lenne, méghozzá a következő: `{8, 15, 32, -10, 0}`. Legtöbbször nem ez a célunk, így ne felejtsük el mindkét dimenzió indexét megadni!

Az egydimenziós tömb elemeinek billentyűzeten való beírását, valamint az elemek listázását `for` ciklus segítségével oldottuk meg. A szituáció itt is hasonló, de itt mivel egy helyett két dimenzió van, ezért két `for` ciklusra lesz szükség, amiket egymásba kell ágyaztatni. Az egyik ciklus a sorokért fog felelni, a másik pedig az oszlopokért. Legtermészetesebb az, ha a kétdimenziós tömb elemein a sorok szerint haladunk végig. Ekkor a külső ciklus fog a sorokért felelni, a belső pedig az oszlopokért. A kétdimenziós tömb elemeinek beírását tehát a következőképp oldhatjuk meg:

```
int tomb[3][5], i, j;
for(i = 0; i < 3; i++) {           // sor lefixálása
    for(j = 0; j < 5; j++) {       // a lefixált sor elemein való
        scanf("%d", &tomb[i][j]);  //   végighaladás
    }
}
```

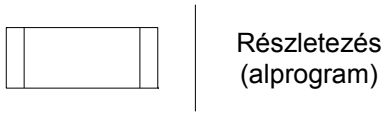
A tömb elemeinek listázása is hasonlóképp működik:

```
for(i = 0; i < 3; i++) {           // a tomb[3][5] tömb listázása
    for(j = 0; j < 5; j++) {
        printf("%d\t", tomb[i][j]);
    }
    printf("\n");       // amikor kilistáztunk az egész sort, a következő sor
}                       //   elemeit listázzuk egy új sorba
```

Függvények

Már nem egyszer említettük, hogy a sikeres programozás egyik alapkövetelménye az, hogy a megoldandó problémát fel tudjuk bontani több kisebb problémára (részproblémákra). Ez logikus is, hiszen az ember sokkal könnyebben birkózik meg a kisebb méretű problémákkal, mint a nagyokkal. Azonban azt is észre lehet venni, hogy ahogy múlik az idő, úgy komplikálódnak a problémák is: a főprobléma nagyobbodik, a részproblémák sokasodnak, és már a részproblémák is kezdenek annyira bonyolultak lenni, hogy őket is fel kell bontani kisebb problémákra. Ezt természetesen a programkódon is észre lehet venni, hiszen hossza egyre növekszik, az áttekinthetőség viszont csökken. Mi lenne, ha az egyes részproblémákat kiszednénk a programunkból, külön helyet biztosítanánk nekik, nevet adnánk nekik és a főprogramból csak hívnánk őket nevük szerint? Ezeket a részproblémákat, amiket kiszedtünk a főprogramunkból, és relatív önállóan léteznek, **alprogramoknak** nevezzük.

Az algoritmusoknál említettünk egy speciális csomópontot, aminek magyarázatára eddig még nem tértünk ki. Ez volt a **részletezés** csomópontja. Lényegében ez a csomópont felel meg az alprogramoknak.



A programnyelvekben kétféle alprogramot ismerünk:

a) **Függvények**, vagy **funkciók** (ang. ***Function***) – nevüket a *matematikai függvényekről* kapták. Tudjuk, hogy a matematikai függvények lényegében *leképezések*, amik bizonyos bemeneti értékeken különféle matematikai műveleteket végeznek és eredményül valamilyen kimeneti értéket adnak vissza. A matematikában a függvények jelölése $f(x)=y$. Itt a függvény neve f , aminek van egy x paramétere. A függvény minden x értékre egy eredményt ad vissza, ez az y . Ezt úgy kapjuk meg, hogy ha a függvénybe behelyettesítjük az x konkrét értékét, kiszámolhatjuk az y értékét. A függvények le is rajzolhatók a koordináta-rendszerben, ekkor az x tengely egy bizonyos értékén megnézhetjük, hogy mennyi lesz az y . A matematikai függvényekre metaforikusan úgy is tekinthetünk, mint valamilyen gépezetre egy **dobozban**, aminek belsejében valamilyen **adatfeldolgozás** történik, és aminek teteje és alja nyitott. A *doboz teteje* azért nyitott, mert a dobozba be szeretnénk dobni valamilyen adatokat (pl. valamilyen x értéket), amiket majd a dobozban található gépezet fel fog dolgozni. A *doboz alja* viszont azért nyitott, mert a feldolgozás után valamilyen eredmény fog születni (y), és szeretnénk, ha ez az adat kiesne a dobozból. Tehát, a doboz tetején keresztül adjuk meg a bemeneti adatokat, a doboz alján pedig a kimeneti adat (eredmény) fog kiesni. Ez a doboz, mint metafora, tökéletes az alprogramok megértésére.

   Tehát, a **függvény** egy olyan gépezet dobozba zárva, aminek **teteje és alja is nyitott**. Vagyis, a függvények a programozásban olyan programkódok, vagy programrészek, amik szintén valamilyen értékeket, pontosabban adatokat fogadnak bemenetként (lehet bármi, számtól egészen tömbig), azokat feldolgozzák, és eredményül valamilyen kimeneti adatot adnak vissza. Fontos megjegyezni, hogy habár a bemeneti adatok száma lehet nulla, egy, vagy több, a kimeneti adatok száma mindig egy, tehát a függvény végrehajtásának **pontosan egy** eredménye lesz.

b) **Eljárások** vagy **procedúrák** (ang. ***Procedure***) – nagyon hasonlítanak a függvényekre, azzal a különbséggel, hogy ezek **alulról zártak**, azaz **nem adnak vissza** eredményül kimeneti adatot. Persze fel lehet tenni akkor a kérdést, hogy mi egyáltalán a feladatuk. Az eljárások **belülről** fejtik ki hatásukat, és így kihatnak a főprogramra is. Úgy is lehet mondani, hogy

míg a függvények a bemeneti adatokat békén hagyják, és helyette egy új kimeneti adatot hoznak létre, addig az eljárások magukat a bemeneti adatokat módosítják tartósan és egyéb más tartós műveleteket hajtanak végre.

Vannak programnyelvek, amik mindkét alprogramtípust támogatják. Viszont vannak olyan programnyelvek is, amik például csak a függvényeket támogatják. A C programozási nyelv is ilyen. Tehát a C programnyelv hivatalosan csak a függvényeket támogatja. Azonban a C-ben a függvények sokkal többre képesek, ugyanis ötvözik nem csak az egyszerű függvényeket, hanem az eljárásokat is. Így lényegében egy C-függvényben akár keverhetjük is a két alprogramtípust, tehát készíthetünk olyan függvényt is, aminek kimeneti értéke is van, de a bemeneti adatokat is módosítja. Vagy készíthetünk tiszta eljárást is, tehát ami nem fog visszatérési értéket generálni.

Alprogramok nélkül nem tudnánk elképzelni a modern programnyelveket, hiszen rettenetesen megkönnyítik a programozást. Az alprogramokat többek között a következő okokból használjuk:

- **Lerövidítik a programot** – tételezzük fel, hogy a programunkban több helyen kell egy kétdimenziós tömböt feltölteni elemekkel. Megtehetjük azt, hogy minden szükséges helyre dupla `for` ciklust helyezünk, vagy megtehetjük azt, hogy a tömb feltöltését alprogramba tesszük, és a főprogramban csak hivatkozunk rá szükség szerint. Ezzel nem csak a főprogram lesz rövidebb, hiszen jó párszor megmenekülünk a dupla `for` ciklusok írásától, hanem lerövidíti az egész programot is, hiszen nem kell többször leírni a tömb feltöltésének algoritmusát, hanem elég egyszer: az alprogramban.
- **Áttekinthetővé teszik a programot** – ha minden részproblémát a főprogramban oldunk meg, csökken a program áttekinthetősége. Azonban ha a részproblémák egy részét alprogramok formájában kivesszük a főprogramból, és csak szükség szerint hivatkozunk rájuk, sokkal áttekinthetőbb lesz a főprogram is, de a részproblémák is. Ugyanis a főprogram nem bombáz minket a részproblémák megoldásával, ha nem ez a célunk, másrészt ha csak az alprogramot nézzük, az csak a részprobléma megoldását tartalmazza.
- **Később is felhasználhatók** – a megírt alprogramok relatív függetlenül élnek a főprogramtól. Ha akarjuk, felhasználhatjuk őket a későbbi programokban is.
- **Csapatmunkára ösztönöznek** – az alprogramok, mivel relatív függetlenek és jól elkülönülnek a főprogramtól, lehetővé teszik, hogy a nagy projekteket több programozó írja párhuzamosan. A főprogramot írhatja az egyik programozó, az egyik alprogramot függetlenül írhatja egy másik programozó, egy másik alprogramot írhatja egy harmadik programozó, stb.

Habár a C-függvények egyesítik a függvények és eljárások erejét, mi azonban jó ideig csak az igazi függvényekkel fogunk foglalkozni (tehát amik nem módosítják a bemeneti adatokat, hanem egy kimeneti adatot adnak vissza), valamint a csonka eljárásokkal (amik nem adnak vissza semmit, de nem is módosítják a bemeneti adatokat).

A függvények definiálása

A függvények létrehozásának szintaxisa nagyban hasonlít a matematikai függvényekre. Vegyük például az $f(x)=x^2$ függvényt. Az f betű a függvény jelölése (azaz *neve*), a zárójelben található x a bemeneti érték (sokszor hívják *argumentumnak* vagy *paraméternek*), az x^2 pedig a függvény *definíciója*, ami azt mondja el, hogy mire való a függvény, hogyan alakítja át a bemeneti értéket. Az $f(x)=x^2$ egy konkrét függvény, és az x változó konkrét számokkal való behelyettesítésével konkrét eredményeket kapunk, pl. $f(2)=4$, vagy $f(4)=16$.

Ahhoz, hogy C-ben definiáljunk egy függvényt, a következőkre van szükség:

- **Függvénynév** – mivel több függvényünk lehet, ezért a függvénynek *azonosítót* (nevet) kell adni. A függvényre később a neve alapján fogunk hivatkozni. A névadás feltételei megegyeznek a C általános névadási feltételeivel, vagyis csak az angol ábécét használhatjuk, a név nem kezdődhet számmal, stb.
- **Paraméterlista** – a bemeneti adatokat kell benne megadni, zárójellel körbevéve. Az egyszerű matematikai függvényekkel ellentétben nem csak egy paramétert definiálhatunk egy függvénynél, hanem **többet** is, de azt is megtehetjük, hogy **egyet sem** definiálunk. Ezenkívül a paraméterek nem csak számok lehetnek. A paraméter típusa lehet bármely alaptípus, sőt még tömb is. Fontos megjegyezni, hogy a függvény definiálásánál nem foglalkozunk a paraméter(ek) konkrét adatokkal való behelyettesítésével, tehát marad az $f(x)$ jelölés, ahol x a paraméter, ezért a függvények definiálásánál használt paramétereket **formális paramétereknek** is nevezzük.
- **Függvénytörzs** (vagy **függvénytest**) – itt magyarázzuk el a C-nek, hogy hogyan fog a függvény működni, lényegében ez a függvény algoritmusa.
- **Visszatérési érték típusa** – a matematikai függvényekkel ellentétben, ahol a függvény eredménye valamilyen számérték, addig a C-ben bármilyen adat lehet. Mint ahogy szükség van egy változó definiálásánál megadni nem csak annak nevét (azonosítóját), hanem típusát is (egész, valós számok, karakterek, stb.), úgy a függvények visszatérési értékének is meg kell adni a típusát. Természetesen a C-függvényeket eljárásként is használhatjuk, vagyis a visszatérési érték léte nem kötelező.

Ezek alapján már megnézhetjük, hogy néz ki C-ben a **függvénydefiníció** szintaxisa:

```
visszatérési_típus függvénynév(paraméterlista) {
    függvénytörzs
}
```

…ahol `visszatérési_típus` a függvény visszatérési értékének (eredményének) típusa, `függvénynév` a függvény neve (azonosítója), a zárójelben található `paraméterlista` a formális paraméterek listája, `függvénytörzs` pedig a függvény algoritmusa, tehát lényegében C-utasítások sorozata. A fenti szintaxisban az első sort, ami a visszatérési típust, a függvénynevet és a paraméterlistát tartalmazza, pár szakirodalomban a **függvény fejlécének** is nevezik.

A fenti szintaxisban található `paraméterlista` a függvény formális paramétereit tartalmazza. Minden paramétert a következő szintaxis szerint adunk meg:

```
típus paraméter_név
```

…ahol `típus` a paraméter típusa (pl. `int`, `float`, `char`, stb.), `paraméter_név` pedig a formális paraméter azonosítója. A paraméter lényegében egy változó, és hasonlóképp definiálandó, mint az egyszerű változó is: a típus és az azonosító megadásával. Azonban a következő dolgokra figyeljünk oda:

- Egy paraméter típusa lehet bármi: alaptípus (`int`, `float`, `char`, stb.), de akár összetett típus is (például tömb).
- Ha több paraméterből áll a paraméterlista, akkor a paramétereket **vesszővel** (`,`) választjuk el egymástól. Azonban az egyszerű változók definiálásával ellentétben itt **tilos** a típus jelölését elhagyni azonos típusú paraméterek esetén. Tételezzük fel, hogy három egész típusú változót (`p`, `q`, `r`) kell létrehozni. Míg ezt a főprogramban megtehettük az `int p, q, r;` definíciós sorral, addig a paraméterek megadásánál ki kell írni minden változónévnél a típust (`int`), még akkor is, ha a típus mindhárom paraméternél ugyanaz. Tehát `(int p, int q, int r)`.
- Mivel formális paraméterekről van szó, ezért a változók definiálásánál tanult változó inicializálása (a változó kezdőértékének megadása, pl. `int q = 2;`) szintén **tilos**. Ez logikus is, hiszen a paraméternek majd csak akkor adunk konkrét értéket, amikor hívni fogjuk a főprogramból.
- Ha egy programon belül több függvényt definiálunk, a paramétereknek adhatjuk ugyanazt a nevet, míg azok más-más függvényben helyezkednek el. Így például használhatjuk az `x` paraméternevet az `f1` és az `f2` függvényben is. Ezt azért tehetjük meg, mert a függvények relatív függetlenek egymástól.

Ne feledjük még a következőket sem:

- Ha a paraméterlista nem tartalmaz egy paramétert sem, akkor is **ki kell tenni** a listát körbevevő **zárójelet**, csak ebben az esetben azt üresen kell hagyni!
- A visszatérési érték típusa bármely alaptípus lehet (`int`, `char`, `float`, stb.), az összetett típusok közül pedig a mutatók, struktúrák és az uniók engedélyezettek. Sajnos a visszatérés értékének típusa nem lehet tömb, de szerencsére erre is van megoldás. Erről később még szó lesz.
- Ha a függvényt **eljárásként** szándékozunk használni, akkor nem lesz neki visszatérési értéke, azaz eredménye. Ebben az esetben a visszatérési típus helyett a **`void`** kulcsszavat kell írni.
- A függvénynek érdemes olyan nevet választani, ami jól jelképezi a függvény feladatát, célját. A változókhoz hasonlóan a függvény nevét illik kis kezdőbetűvel írni.
- Ha egy programon belül több függvényt definiálunk, adjunk nekik különböző nevet!

Mielőtt továbblépünk, írjunk le pár példát függvénydefiniálásra (függvénytörzs nélkül):

```
int negyzet(int x) {                /* jó példa */
    /* függvénytörzs  */
}
```

```
float keplet(int p, q, r) {  /* rossz példa, minden paraméternél ki */
                             /* kell tenni a típust                 */
}


float keplet(int p, int q, int r) {      /* jó példa */

}


void szamokKiirasa(int elso, float masodik) {  /* nincs visszatérési */
                                               /* érték, ezért void  */
}

int felhasznalo {         /* rossz példa, még ha nincs is paraméter, */
                          /* ki kell tenni a zárójeleket             */
}

int felhasznalo() {       /* jó példa, ki van téve a zárójel         */

}
```

A függvénytörzs

A függvénydefiníció szintaxisában említett **függvénytörzs** vagy **függvénytest** lényegében a függvény algoritmusa, a részprobléma megoldásának levezetése. A függvénytörzs alapjában véve egy **utasításblokk**, azaz C-utasítások sorozata. Mint már az elágazásoknál és ciklusoknál említettük, az utasításblokkot **kapcsos zárójelek** (**{** és **}**) közé kell zárni. A függvényeknél sincs ez másképp.

Az utasításblokk nagyon hasonlít a főprogramra, tehát ha szükség van rájuk, új változókat vezethetünk be a blokk elején, majd jöhetnek a különböző utasítások. A függvényben létrehozott új változók **helyi**, azaz **lokális** jellegűek, ami annyit jelent, hogy a függvényen kívül nem léteznek. Amikor lefuttatjuk a főprogramot, a függvényben található változók csak akkor jönnek létre, amikor a főprogram a függvényhívás operátorához ér és elkezdi a függvény futtatását. Amikor a függvény futása befejeződött, megsemmisülnek a függvényben létrehozott változók, és folytatódik a főprogram végrehajtása. Ha a főprogram másodszor is ugyanezt a függvényt hívja, ismét létrejönnek a függvény változói, mintha mi sem történt volna. Ezért mondjuk azt, hogy a függvény változói lokális (helyi) jellegűek: csak akkor és addig élnek, míg fut a függvény. A főprogramhoz hasonlóan a függvénybeli változókat inicializálhatjuk, azaz kezdőértéket is adhatunk nekik.

Amennyiben igazi függvényként használjuk a C-függvényt, tehát a függvénynek lesz visszatérési értéke (eredménye), a függvénytörzsben ezt meg is kell jelölni. A C-ben erre a **`return`** kulcsszavat használjuk, aminek szó szerinti fordítása *„térj vissza”*. Ennek szintaxisa kétféle lehet:

`return kifejezés;` vagy **`return;`**

A szintaxis első változatában a `kifejezés` nem más, mint az a változó, vagy konstans, aminek értékét szeretnénk a függvény eredményének kikiáltani. A kifejezés értékének típusa meg kell, hogy egyezzen a *függvénydefiníció elején* írt visszatérési érték típusával, ha mégsem, a C megpróbálja azt átalakítani. Amikor a program a `return` kulcsszavat tartalmazó programsorhoz ér, kiesik a függvényből és a változó vagy konstans pillanatnyi értékével visszatér a főprogramhoz, méghozzá a függvényhívás utasításához.

Ezzel szemben a második szintaxist, ahol egymaga áll a `return` kulcsszó, akkor használjuk, amikor *eljárásként* használjuk a C-függvényt, tehát a visszatérés típusának a `void` kulcsszót adtuk meg. Jelentése ugyanaz, mint az első szintaxisnál is, tehát megszakad a függvény futása és visszatérünk a főprogramhoz a függvényhívás helyére, a különbség csak az, hogy nem viszünk semmiféle visszatérési értéket magunkkal, hiszen most eljárásról beszélünk.

Érdemes megjegyezni még a következőket:

- Amennyiben a függvényt eljárásként használjuk (a visszatérés típusa `void`), **nem kötelező** kitenni a `return` utasítást. Ebben az esetben a program automatikusan kilép a függvényből, amikor annak utasításblokkjának **végére** ér. Ezzel szemben az igazi függvényként használt függvényeknél **kötelező** a függvényben legalább egyszer használni a `return kifejezés;` utasítást, különben a C nem tudná, milyen változónak, vagy konstansnak az értékét szeretnénk visszahozni a függvény végrehajtása után.
- A `return` utasítást többször is használhatjuk egy függvényen belül, függetlenül attól, hogy igazi függvényről, vagy eljárásról van szó. Ez akkor lehet hasznos, ha például *elágazást* tettünk a függvénytörzsbe, és mindegyik ág más és más értéket ad vissza eredményül. Mivel a `return` utasítás a függvény megszakítását eredményezi (a függvényből való kilépést), ezért ha maga az utasítás nem a függvénytörzs utolsó sorában helyezkedik el, akkor a törzs maradék része már nem fog végrehajtódni.
- Tudjuk, hogy a főprogramban két változó ugyanolyan névvel nem létezhet. Azonban, mivel a függvények relatív független programrészek, ezért két függvényben használhatjuk ugyanazokat a változóneveket, hiszen ezek egymástól függetlenek. Természetesen mindkét függvényben külön kell definiálni a változókat. Így például létrehozhatunk egy egész típusú `y` változót az `f1` és az `f2` függvényben is.

Mielőtt továbblépünk, készítsük el a korábban már említett $f(x)=x^2$ matematikai függvény C-variációját. Legyen a függvény neve f helyett `negyzetreEmeles`, a visszatérési érték pedig legyen `int`. Megjegyeznénk, hogy lényegében ez a probléma talán túl egyszerű ahhoz, hogy külön függvényt készítsünk neki, de példának megteszi:

```
int negyzetreEmeles(int x) {
    int y;            /* helyi y változó létrehozása, csak ezen a  */
    y = x * x;        /* függvényen belül fog létezni              */
    return y;     /* legyen az y pillanatnyi értéke a függvény */
}                 /* eredménye                                 */
```

Könnyű észrevenni, hogy ez a fenti függvény szintaktikailag rettenetesen hasonlít magára a főprogramra: ott is új változókat hozunk létre, különböző utasításokat írunk, stb. Ha a függvény nevét `main`-re írnánk, üresen hagynánk a paraméterlistát és `return y;` helyett azt írnánk, hogy `return 0;`, lényegében a főprogramot kapnánk. A hasonlóság egyébként nem véletlen: a C-ben a függvények annyira fontosak, hogy minden probléma, minden program egy **főfüggvényből**, valamint különböző számú **alfüggvényből** áll. Amit eddig **főprogramként** emlegettünk, az egy

`main()` nevű függvény, paraméterlista nélkül, visszatérési értéke pedig `int`. Mint tudjuk, a főprogram utolsó sora mindig `return 0;` volt, ami azt jelenti, hogy a főprogram a futás után mindig nullát ad vissza eredményül. Ez az eredmény annak szól, aki magát a programot hívta, ez pedig az operációs rendszer: a 0 szám jelzi azt, hogy a program futása sikeresen befejeződött.

A függvényhívás

A függvényeket nem csak definiálni kell, hanem a megfelelő módon tudni is kell őket hívni. Habár eddig mindig úgy említettük, hogy a függvényeket a főprogramból hívjuk, ez egyáltalán nem kötelező. A hívás történhet bárhonnan: a főprogramból, vagy bármilyen más függvényből.

A függvényhívásról már volt szó az operátorok fejezetben. A függvények hívását a **függvényhívó operátorral** érhetjük el, aminek jelölése **`()`**. Ez egy kétoperandusú operátor, az egyik operandus a **függvény neve**, és a **zárójel elé** tesszük, a másik operátor pedig a **paraméterlista**, amit a **zárójelbe** helyezünk. Tehát a szintaxis:

```
függvénynév(paraméterlista)
```

…ahol `paraméterlista` annyi paraméterből kell, hogy álljon, amennyit definiál a hívott függvény. A paramétereket **vesszővel** (`,`) választjuk el egymástól. A következőkre nagyon oda kell figyelni:

- Ha a paraméterlista **üres**, akkor is **ki kell tenni a zárójelet**, nélküle a hívás érvénytelen. Tehát: **`függvénynév()`**
- A paraméterlista paramétereinek **sorrendje** meg kell, hogy egyezzen a függvénydefinícióban megadott paramétersorrenddel. Ez a paraméterek típusára is vonatkozik. Például ha a függvénydefinícióban a paraméterlista a következő: `(int elso, float masodik, char betu)`, akkor a függvényhívás során a zárójelbe három paramétert kell tenni a definícióban megadott sorrendben: az első paraméter `int` típusúnak kell, hogy legyen, a második paraméternek `float` típusúnak, a harmadiknak pedig `char` típusúnak.

Míg a függvénydefinícióban található paraméterlista paraméterei nem tartalmaztak konkrét értékeket (csak a paraméter *típusát* ismertük, valamint *azonosítóját*, azaz nevét), ezért *formális paramétereknek* is neveztük őket, addig a *függvényhívás* paraméterlistája már **konkrét** értékeket kell, hogy tartalmazzon. Ezért ezeket a paramétereket **aktuális paramétereknek** is nevezzük. Ha a matematikai függvényeket hozzuk fel hasonlatként, akkor az $f(x)$ -ben az x egy formális paraméter, hiszen az x értéke lehet bármi, míg az $f(2)$ -ben a 2 már egy konkrét érték, tehát aktuális paraméter. Az aktuális paraméter lehet bármilyen kifejezés, aminek van konkrét értéke (konstans, változó), vagy ki lehet azt számolni (aritmetikai műveletek, egyéb függvények visszatérési értéke, stb.) . Aktuális paraméter így például lehet:

- `3` – **konstans**;
- `x` – egy **változó**, aminek van értéke;
- `x+y` – egy **aritmetikai művelet**, először kiszámolódik az `x+y` értéke, és ez lesz az aktuális paraméter; stb.

Csak ahogy már említettük, vigyázzunk arra, hogy az aktuális paraméter típusa megegyezzen a formális paraméter típusával a függvénydefinícióban.

Mint tudjuk, egy C-függvény viselkedhet igazi függvényként, vagy eljárásként (`void`). Amennyiben a függvényt **eljárásként** használjuk, tehát nem hoz vissza eredményt (`void`), akkor a függvényhívást csak mint **önálló utasítást** használhatjuk (egyedül áll a programsorban). Például ha a függvény definíciója `void fv(int a)`, akkor a hívás például így nézhet ki:

```
/* a program kezdete, k változó létrehozása */
fv(k);
/* a program többi sora */
```

Viszont ha a függvénynek **van visszatérési értéke** (eredménye), akkor ahhoz, hogy az eredmény megmaradjon, nem önálló parancsként kell a hívást írni, hanem **kifejezésként** kell rá tekinteni, és minden olyan helyre írhatjuk, ahova valamilyen kifejezést várunk, aminek konkrét értéke lehet. Például **értékadó operátor jobbértékeként**, vagy akár egy másik **függvény aktuális paramétereként**. Azonban használhatjuk **önállóan** is (mintha eljárás lenne), ekkor viszont a visszatérési érték elveszik. Vegyük például a nemrég definiált `negyzetreEmeles(int x)` függvényt, aminek visszatérési típusa `int`, feladata pedig a paraméter négyzetre való emelése. Pár példa, hogyan lehet a függvényt hívni:

```
x = negyzetreEmeles(2);       /* 1. példa */

negyzetreEmeles(szam);        /* 2. példa */

printf("Az m+n negyzete: %d", negyzetreEmeles(m + n));     /* 3. példa */
```

Vegyük sorjában a három példát:

1. Az **első példa** érthető: az `x` változó értéke legyen a `2` konstans négyzete. Ekkor először az értékadó operátor (`=`) jobbértéke (tehát a függvény) számolódik ki, a függvény eredményül `4`-t ad vissza, tehát ez lesz az `x` értéke.
2. A **második példa** is hasonló, a `szam` nevű változó mostani értékét szeretnénk négyzetre emelni (tételezzük fel, hogy a program elején kapott valamilyen értéket). Viszont érdekes, hogy a függvényhívás most egyedül áll, így az eredmény el fog veszni. Vagyis hiába számolta ki a függvény a `szam` változó négyzetét, és hiába adta azt vissza eredményként, ez az érték el fog veszni, mert nem rendeltük hozzá egy változóhoz sem (például az első példával ellentétben, ahol az `x` változó átveszi a függvény eredményét). Legtöbbször nem ez a célunk, így vigyázzunk erre.
3. A **harmadik példa** viszont igen érdekes. Itt a `negyzetreEmeles()` függvényt egy másik függvény (`printf()`) paramétereként használjuk. Jól tudjuk, hogy a `printf()` is egy függvény, méghozzá egy könyvtári függvény, feladata az adatkiírás. Azt is tudjuk, hogy a `printf()` legtöbb esetben két paramétert tartalmaz: az első a formátum, amit idézőjelbe kell tenni, a második paraméter pedig az argumentumlista, ami röviden szólva annyi elemből áll, ahány `%` jel van a formátumban. Azonban ahhoz, hogy a `printf()` függvényt futtatni lehessen, először minden belső paraméterét ki kell értékelni. Mint látjuk, a *második* paraméter (az argumentumlista) egy *összetett* kifejezés, ebben az esetben egy *függvényhívás*, így először azt fogja a C lefuttatni, majd amikor megjön annak eredménye, csak akkor fog hozzá a `printf()` végrehajtásához. Más szóval, ahhoz, hogy a `printf()` függvényt le

lehessen futtatni, először ki kell számolni a `negyzetreEmeles(m + n)` értékét, és csak aztán fog a `printf()` végrehajtódni.

A függvények programban való elhelyezése

Most már tudjuk, hogyan kell definiálni (létrehozni) függvényeket, és hogyan kell őket hívni. Az utolsó dolog, amiről nem volt még szó, az a függvénydefiníciók *elhelyezése* a programban. A függvényeket többféleképp lehet elhelyezni. A nagyon gyakran használt függvényeket, amiket akár több programon belül is használnánk, érdemes felhasználói függvénykönyvtárba helyezni. Mi lényegében így használtuk a `printf()` és `scanf()` függvényeket, amiket be kellett olvasni a `stdio.h` állományból az `#include` direktívával. Tehát tetszőleges névvel létrehozhatjuk a saját könyvtárunkat is, és behelyezhetjük a függvényeinket.

Mi viszont megelégszünk a legegyszerűbb függvénytárolási móddal: a függvényeket a **főprogram állományába** fogjuk helyezni. Ez azt jelenti, hogy a `.C` kiterjesztésű fájl a `main()` főprogramon kívül még egy vagy több függvény definícióját is tartalmazni fogja, pl. a `negyzetreEmeles` függvényt. A kérdés csak az, hogy *hova* tegyük ezeket a függvénydefiníciókat: a `main()` föle, vagy alá. A rövid válasz erre az, hogy a `main()` fölé. Ugyanis a C-ben csak olyan függvényeket lehet hívni, amik definíciója a **hívás előtt** (azaz **fölött**) található. Ez lényegében azt jelenti, hogy a `negyzetreEmeles()` függvény definícióját a `main()` függvény fölé kell tenni. Ha nem így tennénk, azaz a függvényt a hívás alá helyeznénk, implementációtól függően vagy hibaüzenetet kapnánk, vagy a C megpróbálná magától „kitalálni” a függvény fejlécét. Hasonlóképp, ha a `negyzetreEmeles()` függvény a függvénytörzsben egy másik függvényt is hívna, akkor azt a `negyzetreEmeles()` függvény *fölé* kellene tenni. Vannak a C-nek olyan implementációi, amik nem ilyen szigorúak, de a kockázat elkerülése érdekében jobb ezt a szabályt betartani.

Tehát a következő példa a helyes:

```
#include <stdio.h>

int negyzetreEmeles(int x) {    /* a függvény hívása a main()-ből jön */
    int y;
    y = x * x;
    return y;
}

int main()
{
    int szam, negyzet;
    scanf("%d", &szam);
    negyzet = negyzetreEmeles(szam);     /* a függvény a main() fölött van */
    printf("A %d negyzete: %d", szam, negyzet);
    return 0;
}
```

Azonban a függvények száma egy projektumban rengeteg lehet, és a függvények módosításaival akár a függvények fizikai sorrendjén is változtatni kellene. Ennek elkerülése érdekében lettek a függvényprototípusok kitalálva.

A függvények prototípusa (haladó szint)

A **függvényprototípusok** lényegében **függvénydeklarációk**, bár a szakirodalom különbséget tesz a kettő között. Tehát, míg az eddig emlegetett függvénydefiníció az egész függvényt tartalmazza (a fejlécet és a függvénytörzset is), addig a prototípus (vagy deklaráció) csak a fejlécet, vagyis a függvény leírását: annak nevét, paraméterlistáját és visszatérési értékének típusát. A prototípus nem kíváncsi a függvény működésére (függvénytörzsére). Lényegében a függvény fejléce (első sora) a fontos. A prototípusok írása egyszerű, a szokásos függvénydefiníciótól csak két dologban különböznek:

- A kapcsos zárójelekbe helyezett függvénytörzs *helyett* csak egy **pontosvesszőt** (`;`) kell tenni;
- Amennyiben a paraméterlista a függvénydefinícióban *üres*, akkor az üres zárójel, tehát `()` írása *helyett* a zárójelbe a **`void`** kulcsszavat kell tenni, pl. `float fuggv(void);`

A prototípusokat illik a **program elejére** tenni, rögtön az előfeldolgozó utasításai után. Mivel megadtuk már a programunk „tetején” a függvények prototípusát, ezért a C számára most már ismertek, így a függvény igazi definícióját lényegében tehetjük a program bármely részébe, akár a `main()` *alá* is, ha nekünk az úgy jobban tetszik. Ezzel sok fejfájástól kímélhetjük meg magunkat, amennyiben sok függvényt definiálunk a programunkban.

Sok szakirodalom melegen ajánlja a prototípusok használatát, még akkor is, amikor a függvények száma kevés a programban. Azonban ne feledjük: attól, hogy a program tartalmazza a függvény prototípusát, még nem azt jelenti, hogy a függvényt nem kell a program valamelyik későbbi pontján *definiálni is*.

Demonstrációként írjuk le még egyszer az előző programot, de most prototípus használatával:

```
#include <stdio.h>

int negyzetreEmeles(int x);    /* a prototípus */

int main()
{
    int szam, negyzet;
    scanf("%d", &szam);
    negyzet = negyzetreEmeles(szam);
    printf("A %d negyzete: %d", szam, negyzet);
    return 0;
}

int negyzetreEmeles(int x) {  /* a függvénydefiníció most main() alatt van, */
    int y;                    /* de nem gond, mivel létezik prototípus      */
    y = x * x;
    return y;
}
```

A paraméterátadás (haladó szint)

Egy fontos dologról még nem beszéltünk. Tételezzük fel, hogy egy függvényhívás aktuális paramétere egy változó. Fel lehet tenni a kérdést, hogy *mi történne* akkor, ha függvény törzsében módosítanánk az átadott paraméter értékét? A főprogramba (vagy a hívó függvénybe) való visszatéréskor a paraméterként használt változó értéke szintén *megváltozna*, vagy maradna a *régi*? Ennek demonstrálására nézzük meg a következő programot:

```
#include <stdio.h>

void kiir(int n);

int main()
{
    int szam = 5;
    printf("Hivas elott: %d\n", szam);
    kiir(szam);
    printf("Hivas utan: %d\n", szam);
    return 0;
}

void kiir(int n)
{
    printf("Fuggvenyen belul: %d\n", n);
    n++;
    printf("Fuggvenyen belul: %d\n", n);
}
```

Futassuk le a fenti programot. A következő kimenetet fogjuk kapni:

```
Hivas elott: 5
Fuggvenyen belul: 5
Fuggvenyen belul: 6
Hivas utan: 5
```

Vagyis a válasz a fenti kérdésre az, hogy a függvényen belül akárhogy módosítjuk a paraméterek értékét, azok a függvényből való visszatéréskor visszakapják eredeti értéküket. Alapjában véve arról van szó, hogy a C függvényhívás során nem konkrétan a változókat adja át a függvénynek paraméterként, hanem azok **értékét**. Függvényhívás során a C **helyi (lokális) másolatokat** készít a formális paramétereknek, amik értékei felveszik függvényhívás során az aktuális paraméterek értékeit. Tehát a fenti példában a `kiir(szam)` függvényhívás nem a `szam` változót küldi el a függvénynek, helyette készül róla egy másolat (`int` típusú, ahogy a függvénydefiníció megköveteli), ami felveszi a `szam` változó értékét (ami `5`). A függvény futása közben ez a paraméter `6`-ra növekszik. Azonban amikor visszatérünk a függvényből, a másolat (a függvénnyel együtt) megsemmisül, így a `szam` változónak marad a régi értéke (ami `5`). Ne feledjük: ez a féle viselkedés nem azért van, mert a függvényhívásban használt paraméter neve (`szam`) eltér a függvénydefinícióban használt paraméter nevétől (`n`). A függvénydefinícióban a paramétert elnevezhettük volna pont úgy, ahogy a hívásban is van (`szam`), akkor is ugyanezt a viselkedést kaptuk volna. A válasz abban rejlik, hogy a C-ben a **paraméterek átadása** alapértelmezetten **érték**

szerint történik, tehát az aktuális paraméterekről másolatok készülnek, és a függvény azokkal dolgozik.

Létezik a **paraméterátadásnak** egy másik módja, ez a **cím szerinti átadás**. Ehhez azonban szükséges a **mutatók** (ang. ***Pointer***) ismerete, így ezzel az átadási móddal nemigen fogunk foglalkozni. Elég annyit tudni, hogy a cím szerinti átadásnál nem készül helyi másolat a paraméterről, hanem magát az eredeti változót adja át a fordító a függvénynek, pontosabban annak memóriabeli címét.

Az ún. **skalár** (azaz **alaptípusú**) **paraméterek** mind **érték szerint** lesznek átadva. Azonban a **tömbök** kivételt képeznek, ugyanis azok **cím szerint** lesznek átadva, vagyis a tömbök függvényen belüli módosítása tartósan megmarad a függvényből való kilépéskor. Most erről fogunk kicsit részletesebben beszélni.

A függvények és a tömbök (haladó szint)

Mint tudjuk, a függvény paramétere lehet tömb is. Azonban, ahogy nemrég említettük, a tömbök nem érték szerint adódnak át a függvénynek, hanem *cím szerint*. Ez lényegében annyit jelent, hogy függvényhíváskor a tömbből nem fog másolat készülni, amivel aztán a függvény szabadon dolgozhat, hanem az eredeti tömbbel fog dolgozni. Ez azt vonja maga után, hogy a tömb bárminemű módosítása a függvényen belül tartósan megmarad – akár a függvényhívás után is.

Ha tömböket használunk függvény paramétereként, pár dologra viszont oda kell figyelni:

- Az *aktuális* paraméter (a függvényhíváskor használt paraméter) csak a tömb nevét tartalmazhatja, a szögletes zárójelek (`[` és `]`) használata **tilos**. Tételezzük fel, hogy létrehoztunk egy egész típusú tömböt, `tomb` néven. Ekkor, ha a függvény neve `modosit`, akkor egy függvényhívás így kell, hogy kinézzen: `modosit(tomb)` .
- A *formális* paraméter (a függvénydefinícióban) nem csak a tömb nevét kell, hogy tartalmazza, hanem a szögletes zárójeleket (`[` és `]`) is. Azonban a szögletes zárójeleket **üresen** kell hagyni. Folytatva az előbb elkezdett példát, a függvénydefiníció fejléce akkor például így nézne ki: `void modosit(int tomb[])` .
- A figyelmes olvasók valószínűleg most észrevették, hogy az előző kifejtéssel van egy gond: ha sem az aktuális, sem a formális paraméter nem jelzi a függvénnyel a tömb hosszát, honnan fogja a függvény tudni, hogy hány elemből áll a tömb? A válasz egyszerűen: **sehogy**, a függvény egyáltalán nem fogja tudni, hány elemből áll a tömb. Így fennáll annak a veszélye, hogy a tömb használatakor átlépünk az utolsó elemen, ami mindig fatális hibához vezet. Ez miatt kénytelenek leszünk a függvényhez egy új paramétert csatolni: a **tömb méretét**. Megemlítenénk, hogy nem kötelező a tömb maximális méretét átadni paraméterként, elég csak azt a méretet, amit aktívan ki is használtunk. Így ha a tömb mérete egyébként 100, de mi csak az első 20 elemet használtuk benne, akkor a függvény második paramétereként nyugodtan írjunk `20`-t.

Demonstráljuk a tömbök paraméterként való átadását egy példán keresztül:

```
#include <stdio.h>
#define MERET 5

void modosit(int tomb[], int m);

int main()
```

```
{
    int tomb[MERET] = {1, 1, 1, 1, 1};
    int i;
    printf("Hivas elott: ");
    for(i = 0; i < MERET; i++) {
        printf("%d ", tomb[i]);
    }
    modosit(tomb, MERET);
    printf("\nHivas utan:  ");
    for(i = 0; i < MERET; i++) {
        printf("%d ", tomb[i]);
    }
    return 0;
}

void modosit(int tomb[], int m)
{
    int i;
    for(i = 0; i < m; i++) {
        tomb[i] = 2;
    }
}
```

Ez a feladat hasonlít az előzőhöz, tehát kiíratja a függvény paraméterének értékét a függvényhívás előtt és után. A főfüggvény egy `tomb` nevű egész típusú függvényt definiál, amit inicializál is, csupa egyesekkel. Azt is láthatjuk, hogy a `modosit()` függvény második paramétere egy egyszerű `int` típusú változó: ezen keresztül fogjuk átadni a tömb méretét. A `modosit()` függvény egyedüli feladata az, hogy a tömb minden elemét `2`-re írja. Futtassuk le a programot, és a következőket fogjuk kapni:

```
Hivas elott: 1 1 1 1 1
Hivas utan:  2 2 2 2 2
```

Tehát, mint látjuk, a függvényhívás előtt a függvény minden eleme `1`, azonban az elemek értéke átíródik `2`-re a `modosit()` függvényben. Viszont a függvényből való kilépéskor a függvény elemei nem `1`-t fognak tartalmazni, hanem `2`-t. Ez azért van, mert az egyszerű típusokkal ellentétben a tömbök nem érték, hanem *cím szerint* adódnak át a függvénynek.

Mint már korábbról tudjuk, a függvény visszatérési értéke nem lehet tömb. Azonban erre igazság szerint nincs is valami nagy szükség, hiszen a tömbök cím szerint adódnak át a függvénynek, tehát tartósan módosíthatók a függvénytörzsben. Azonban ez a megkötés is kikerülhető mutatók használatával, de mi nem fogunk ezzel külön foglalkozni.

Kiemelnénk azt is, hogy a *többdimenziós* tömbök (így a kétdimenziósok is) új problémákat vetnek fel a függvényekkel kombinálva. A többdimenziós tömbök és a függvények kombinációjával nem fogunk foglalkozni.

A globális változók (haladó szint)

A függvények egy negatív tulajdonsága az, hogy nem képesek több eredményt visszahozni, csak egyet. Így lényegében lehetetlen lenne például egy olyan függvényt írni, ami bemenetként egy téglalap két hosszát kéri és kimenetként visszaadja annak kerületét és területét. Szükség volna a függvény szétszedésére *két kisebb* függvényre. Erre persze van egy nagyon elegáns megoldás, ez pedig az, hogy küldjünk el bemenetként *még két paramétert* – a téglalap kerületét és területét, de ezt a két paramétert ne érték, hanem *cím szerint* adjuk át. Ekkor ha a függvényen belül megváltoztatjuk a két paraméter értékét, a módosítás a függvényen *kívül is* látszódni fog. Azonban ehhez nélkülözhetetlen a *mutatók* (ang. *Pointer*) ismerete, amikkel mi nem foglalkozunk.

Ehelyett egy *alternatív megoldást* fogunk megnézni, amivel valamilyen módon megoldható a fenti probléma. Említettük, hogy a függvényekben a változók *helyi*, azaz *lokális* jellegűek, ami annyit jelent, hogy csak függvényen belül láthatók és élnek. Ez alól a főfüggvény, a `main()` sem kivétel, így az ott létrehozott változók sem fognak látszódni az általa hívott függvényekben. Azonban nem mindig ez a célunk. Ha létrehozhatnánk olyan változókat is, amik az **egész programban** (a `main()` főfüggvényben, és az összes többi függvényben is) látszódnának, akkor meg tudnánk oldani a fenti problémát. Ezeket a változókat globális változóknak nevezzük.

A **globális változókat** a helyiekkel ellentétben nem a függvényeken belül definiáljuk, hanem azokon **kívül**. A globális változók a definiálás pozíciójától *kezdve* élnek. Így ha a *program elejére* tesszük őket (rögtön az előfeldolgozó direktívái után), akkor lényegében az egész programban látszódni fognak. Azonban ha két függvény *közé* tesszük őket, akkor csak a definiálás *alatti* függvényeken belül lesznek láthatók, a definiálás fölötti függvényekben nem. Mivel legtöbbször nem ez a célunk, ezért tegyük őket rögtön az `#include` és `#define` direktívák után!

A globális változók használatával megoldódna a fenti téglalapos probléma is. Ugyanis ha a téglalap kerületét és területét globális változók formájában definiálnánk, akkor azok értékét ki tudnánk számolni függvényen belül úgy, hogy azok értéke látszódna a főfüggvényben is. Oldjuk meg tehát ezt a problémát globális változók segítségével:

```
#include <stdio.h>

float a, b, K, T;

void teglalap()
{
    K = (2 * a) + (2 * b);
    T = a * b;
}

int main()
{
    printf("Irja be a teglalap ket oldalat vesszovel elvalasztva: ");
    scanf("%f, %f", &a, &b);
    teglalap();
    printf("A teglalap kerulete %.2f, terulete pedig %.2f", K, T);
    return 0;
}
```

Tehát, mint látjuk, ez egy alternatív megoldás az eredeti problémára, hiszen lényegében nem a paramétereken keresztül oldottuk meg a problémát, hanem egy teljesen más úton.

Globális változók használatakor vigyázzunk, hogy keveredés elkerülése érdekében *ne definiáljunk azon a néven* lokális változókat is, különben nem kívánt eredményeket kapnánk.

A rekurzív függvények (haladó szint)

A **rekurzív függvények** olyan függvények, amik közvetlenül vagy közvetve **saját magukat hívják**. A rekurzív függvények lehetővé teszik olyan problémák elegáns és relatív könnyű megoldását, amik természetüknél fogva rekurzívak.

A rekurzió megértéséhez felhasználhatjuk az egyik gyakran használt matematikai művelet, a *faktoriális* (*n!*) kiszámolását. Lényegében az *n* szám faktoriálisa az 1-től *n*-ig terjedő pozitív egész számok szorzata. Tehát:

n	n!
1	1
2	$1\cdot 2=2$
3	$1\cdot 2\cdot 3=6$
4	$1\cdot 2\cdot 3\cdot 4=24$
5	$1\cdot 2\cdot 3\cdot 4\cdot 5=120$
…	…
n	$1\cdot 2\cdot ...\cdot (n-1)\cdot n$

Még csak az a kérdés, hogy mennyi a *nulla* szám faktoriálisa? Megegyezés szerint az eredmény 1, tehát:

$$0!=1$$

Ezt a feladatot **akkumuláció** segítségével oldottuk meg. **Akkumulálni** annyit jelent, hogy kiválasztunk egy változót, és azt *„akkumuláljuk”*, tehát **saját magát** használjuk fel **operandusként** egy aritmetikai műveletben. Miért? Mert ahhoz, hogy például kiszámoljuk a 4 faktoriálisát, ki kell hozzá számolni az *előző számok* faktoriálisát is. Nézzük meg figyelmesebben a fenti táblázatot. Az 1 faktoriálisának kiszámolása könnyű. Hogyan kapjuk meg a 2 faktoriálisát? Fogjuk az előző szám faktoriálisának eredményét (ami 1), és megszorozzuk 2-vel (az eredmény most 2 lesz). A 3 faktoriálisát úgy kapjuk meg, hogy fogjuk az előzőleg kiszámolt eredményt (ami 2) és megszorozzuk 3-mal (az eredmény most 6). A 4 faktoriálisát úgy számoljuk ki, hogy fogjuk az előző eredményt (ami 6), megszorozzuk 4-gyel, és most az eredmény 24. Vagyis érvényes:

n	n!
1	1 , pontosabban $1\cdot 0!=1$
2	$2\cdot 1!=2$
3	$3\cdot 2!=6$
4	$4\cdot 3!=24$
5	$5\cdot 4!=120$
…	…
n	$n\cdot (n-1)!$

Tehát mindig az előző eredményt használtuk fel az egyik operandusként, hogy egy új eredményt kapjunk. Ezt jelenti az **akkumuláció**, azt a változót, amivel pedig végrehajtjuk az akkumulálást, **akkumulátornak** nevezzük.

A matematikában egyébként a faktoriálist a következőképp definiálják:

$$n! = \begin{cases} n \cdot (n-1)! & , \text{ha } n > 0 \\ 1 & , \text{ha } n = 0 \end{cases}$$

Ezt eddig a ciklusos struktúrák segítségével oldottuk meg, ugyanis az akkumuláció természetes követője a ciklus. Ha megnézzük a fenti táblázatot, érezzük, hogy a ciklusban az 1-től *n*-ig terjedő számok a ciklusváltozót (`i`) jelképezik, az előző szám faktoriálisa az akkumulátor *régi* eredménye, és ha ezt összeszorozzuk a ciklusváltozóval, megkapjuk az *új szám* faktoriálisát (ez lesz az akkumulátor *új* értéke). Vagyis most már fel lehet írni a képletet:

```
fakt = fakt * i;
```

…ahol `fakt` az akkumulátor (jobbértékben a *régi* eredmény, balértékben az *új* eredmény), `i` pedig a ciklusváltozó. Tételezzük fel, hogy létre kell hozni egy függvényt, és benne ki kell számolni a paraméter faktoriálisát. A megoldás a következő lenne:

```
int faktorialis(int n)
{
    int fakt, i;        /* fakt = akkumulátor, i = ciklusváltozó */
    if(n > 0) {
        fakt = 1;       /* inicializáljuk az akkumulátort, ez alapesetben 1 */
        for(i = 1; i <= n; i++) {
            fakt = fakt * i;        /* a képlet */
        }
        return fakt;
    }
    else if(n == 0) {
        return 1;
    }
}
```

Azonban eddig még nem is használtuk a rekurziót, hiszen a függvény nem hívja saját magát. Helyette **iterációval (ciklussal) és akkumulációval** oldottuk meg a problémát. Most meg fogjuk mutatni a **rekurzív** megoldást is. Nézzük meg még egyszer jól a matematikai képletet:

$$n! = \begin{cases} n \cdot (n-1)! & , \text{ha } n > 0 \\ 1 & , \text{ha } n = 0 \end{cases}$$

Látjuk, hogy az *n!* kiszámolásához szükséges az *(n – 1)* faktoriálisának kiszámolása. Vagyis létrehozhatnánk egy olyan függvényt, ami saját magát hívná, de most *(n – 1)* paraméterrel. Az *(n – 1)* paraméterrel hívott függvény tovább hívná magát *(n – 2)* paraméterrel, és így tovább *lefelé*. A önhívogatás addig ismétlődne eggyel kisebb paraméterrel, míg a paraméter értéke *nulla* nem lenne. Ekkor azt mondanánk, hogy „most ne hívd magad, hanem legyen az eredményed 1”. Most megindulna a *felfelé* való haladás, mivel megvan az előző szám faktoriálisa, és amikor a tetejére érkeznénk, megkapnánk a végeredményt.

Tehát látjuk, hogy a rekurzió segédpillére most nem a ciklus (mint az iteratív megoldásnál), hanem az **elágazás**. Az elágazás segítségével fogjuk egyszer *leállítani* az önhívogatást. Ez a

faktoriális kiszámolásának problémájában akkor következik be, amikor *n* értéke egyszer *nulla* lesz. Ha ez nem létezne, **végtelen rekurziót** kapnánk, tehát elméletileg mindörökké ismétlődne a rekurzív hívás, soha nem fejeződne be. Erre ezért fokozottan figyelni kell!

A C támogatja a rekurzív függvényeket és most be fogjuk mutatni a faktoriális kiszámolásának C-változatát rekurzív módon:

```
int faktorialis(int n)
{
    if(n == 0) {
        return 1;
    }
    else {
        return (n * faktorialis(n - 1));       /* rekurzív hívás (n-1)-gyel */
    }
}
```

Mint látjuk, a rekurzív megoldás lényegében direkt veszi a matematikai megoldást. Most a végén hasonlítsuk még össze az iteratív és a rekurzív megoldásokat. Jól látható, hogy a rekurzív megoldás rövidebb, egyszerűbb, elegánsabb, az emberi gondolkodáshoz talán közelibb. Ezzel szemben az iteratív (ciklusos) megoldás akkumulátor segítségével elég barátságtalan, hiszen igen hosszú levezetést igényel, mire az ember rájön a képletre, amit a ciklusba kell helyeznie. Ez miatt a rekurzió igen népszerű, és vannak olyan programozási paradigmák, amik a rekurzióra épülnek.

Azonban a rekurzió nagyobb terhet ró magára a programra. Míg az iteratív megoldásnál egy függvénnyel beérjük (amiben egy ciklus forog), addig a rekurzív megoldásnál nincs ciklus, de cserébe rengeteg függvényhívás (és ezzel függvény-létrehozás) történik. Ez relatív nagy terhet ró a számítógép memóriájára, tehát a végrehajtás hatékonysága csökken.

Ez miatt a programozónak döntést kell hoznia, hogy egy bizonyos problémát iteráció, vagy rekurzió segítségével fog megoldani. A rekurzív megoldás sokszor könnyebb, rövidebb, elegánsabb, az iteratív megoldás viszont hatékonyabb, de nehezebb a programozása.

Felhasznált szakirodalom

Ranđelović, M & Tošić, Ž 2008, Programiranje za II razred elektrotehničke škole, Treće izdanje, Zavod za udžbenike, Beograd

Kraus, L 2004, Programiranje za III razred elektrotehničke škole, Prvo izdanje, Zavod za udžbenike, Beograd

Benkő, T, Benkő, L & Tóth, B 1998, Programozzunk C nyelven!, Változatlan utánnyomás, ComputerBooks, Budapest