Miskolci Egyetem
Általános Informatikai Tanszék

Bevezetés a C programozási nyelvbe

Az Általános Informatikai Tanszék C nyelvi kódolási szabványa

Oktatási segédletek a műszaki informatikus hallgatók számára

Készítette:
Ficsor Lajos

Miskolc
2003

1. Bevezetés

Ez az oktatási segédlet a Miskolci Egyetem műszaki informatikus hallgatói részére készült, a Software fejlesztés I. című tárgy elsajátításának megkönnyítésére.

A jegyzet feltételezi, hogy olvasója rendelkezik alapvető programozástechnikai alapismeretekkel (alapvető vezérlési szerkezetek, típusok stb.), így célja nem programozástechnikai bevezetés, hanem a C programozási nyelv áttekintése. Ugyanakkor az egyes C nyelvi szerkezetek használatát igyekszik példákkal szemléltetni, amelyek egyben az alapvető algoritmusok ismeretének felelevenítését is segítik.

A jegyzet alapja a szabványos (ANSI C) nyelv. Terjedelmi okok miatt nem törekedhettünk a teljességre már az egyes nyelvi elemek ismertetésénél sem, főleg pedig a nyelv használatához egyébként alapvetően szükséges szabványos függvénykönyvtár esetén.

A jegyzetet példaprogramok egészítik ki. A példaprogramok között megtalálható minden olyan kód, amely a szövegben szerepel, de találunk további példákat is.

2. A C programozási nyelv története

A C programozási nyelvet eredetileg a UNIX operációs rendszer részére fejlesztette ki Dennis M. Ritchie 1972-ben, az AT&T Bell Laboratories-ben, részben a UNIX rendszerek fő programozási nyelvének szánva, részben magának az operációs rendszernek és segédprogramjainak a megírására használva. Az idők során ezen a szerepen messze túlnőve kedvelt általános célú programozási nyelvvé vált.

Brian W. **Kernighan** és Dennis M. **Ritchie** 1978-ban adták ki a *The C programming Language* (A C programozási nyelv) című könyvüket, amely hamarosan a nyelv kvázi szabványává vált. Valamennyi fordítóprogram írója - és ezért valamennyi programozó is - ezt tekintette a nyelv definíciójának. Ez volt a **"K&R C"**. A nyelv folyamatos használatának tapasztalatait is figyelembe véve az ANSI (az amerikai szabványügyi hivatal) 1989-ben hivatalosan is szabványosította a C nyelvet. Ezt hívjuk **ANSI C**-nek. Ma már valamennyi fordítóprogramnak ezzel a szabvánnyal összhangban kell készülnie. A jegyzet további részében **C programozási nyelv** alatt mindig az ANSI C-t értjük.

A nyelv elterjedését mutatja, hogy ma valamennyi elterjedt hardware-re (mikrogépektől a nagy main-frame rendszerekig) és operációs rendszer alá elkészítették a fordítóprogramját, az esetek többségében többet is.

3. A C nyelv alapvető tulajdonságai:

1. Viszonylag alacsony szintű, ami az alábbiakat jelenti
   - egyszerű alapobjektumok: karakterek, számok, címek
   - nincsenek összetett objektumokat kezelő utasítások
   - nincsenek input-output utasítások
   - a fentiekből következően viszonylag kicsi, és könnyen elsajátítható
   - végül ezek miatt kicsi és hatékony fordítóprogram készíthető hozzá.

2. Nem támogatja a párhuzamos feldolgozást, a többprocesszoros rendszereket.

3. A C nyelven megírt program általában elég kicsi és hatékony gépi kódot eredményez, ezáltal az assembly szintű programozást a legtöbb területen képes kiváltani.

4. Az egész nyelv logikája és gondolkodásmódja a gépfüggetlenségre törekvésen alapul, ezáltal elősegíti a portábilis programok készítését.

5. Számos implementáció (fordítóprogram, fejlesztési környezet) áll rendelkezésre.

6. A hatékony programozást minden fejlesztési környezetben gazdag függvénykészlet segíti. A rendelkezésre álló függvények az alábbi fő csoportra oszthatók:
   - minden fejlesztési környezetben rendelkezésre álló szabványos függvények
   - adott alkalmazási terület speciális feladatainak megoldását segítő függvények
   - gépspecifikus, de az adott hardware speciális kezelését is lehetővé tevő függvények

A fenti függvények használata - a feladat jellegétől függően - lehetővé teszi a gépfüggetlen programozást és az adott hardware sajátosságait maximálisan kihasználó - és ezért nehezen hordozható - programok írását is.

4. A C nyelvű program felépítése

Minden C program az alábbi alapvető felépítést mutatja:

```
preprocesszor direktívák, pragmák
globális deklarációk

main()
{
lokális deklarációk
utasítások}
függvény - definiciók
```

5. Szintaktikai egységek

A C programok az alábbi alapvető szintaktikai egységekből épülnek fel:

- azonosítók
- kulcsszavak
- állandók
- karakterláncok
- operátorok
- egyéb elválasztók
- megjegyzések

5.1 Azonosítók

Betűk és számjegyek sorozata, betűvel vagy _ (aláhúzás) karakterrel kell kezdődnie. A nagy- és kisbetűk különbözőek. Az azonosítók tetszőleges hosszúságúak lehetnek, de implementációtól függően csak az első meghatározott számú karakterük vesz részt a megkülönböztetésben. (Például a BORLAND C++ 3.1 esetén alapértelmezésben az első 32 karakter szignifikáns. Ez az érték változtatható.)

5.2 Kulcsszavak

Csak meghatározott célra használható azonosítók. Kisbetűvel kell írni. (Tehát **int** kulcsszó, **INT** vagy **Int** általános célú azonosító - bár nem illik használni.)

A hivatkozási nyelv az alábbi azonosítókat tartja fenn kulcsszónak:

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

Az egyes implementációk még újabb azonosítókat is kezelhetnek fenntartott kulcsszavakként.

5.3 Állandók

5.3.1 Egész állandók

A számjegyek sorozatát tartalmazó, nem nullával kezdődő egész szám decimális állandó. Az egész állandó megadható még az alábbi alakokban:

- oktális, ha vezető 0-val kezdődik
- hexadecimális, ha `0X` vagy `0x` vezeti be. A 10-15-ig terjedő számjegyek jelölésére az **`a-f`** vagy **`A-F`** karakterek használhatók.

Ha a decimális állandó értéke az adott implementáció előjeles "`int`" típusának ábrázolási tartományán, vagy ha az oktális illetve hexadecimális állandó az "`unsigned int`" típus ábrázolási tartományán kívül esik, az állandó automatikusan "`long`" típusúvá alakul. (Részletesebben lásd az alaptípusok ismertetésénél.) Ha az egész állandót **`l`** vagy **`L`** betű követi, mindenképpen "`long`"- nak megfelelő helyet foglal el.

5.3.2 Karakterállandó

Aposztrófok között egy karakter. Értéke az adott karakter kódja.

Bizonyos, nem nyomtatható karakterek helyett írható úgynevezett "Escape -szekvencia", az alábbiak szerint:

`\n`	újsor(LF)
`\t`	vízszintes tabulátor
`\v`	függőleges tabulátor
`\b`	backspace (BS)
`\r`	kocsivissza (CR)
`\f`	lapdobás (FF)
`\a`	hangjelzés (BELL)
`\\`	backslash (\)
`\?`	kérdőjel
`\'`	aposztróf

`\"`	idézőjel
`\ooo`	az oktális ooo kódú karakter (ooo egy-, két- vagy háromjegyű oktális szám)
`\xhh`	a hexadecimális hh kódú karakter

Ha a \ karaktert nem a fentiek valamelyike követi, a \ figyelmen kívül marad.

5.3.3 Lebegőpontos állandó

Az

```
egészrész.törtrész E kitevő
```

vagy

```
egészrész.törtrész e kitevő
```

alakú konstans lebegőpontos számot jelent.

Nem hiányozhat egyszerre az egészrész és a törtrész. A tizedespont vagy a kitevőrész közül az egyik elmaradhat, de a kettő egyszerre nem. A kitevőrészben az e vagy E betű és az azt követő szám csak együtt hiányozhat. Utótagként megadható az `f` vagy `F`, amely egyszeres pontosságot, illetve az `l` vagy `L`, amely dupla pontosságot ír elő. Utótag hiányában duplapontosságú.

5.3.4 Karakterlánc

Idézőjelek közé zárt karaktersorozat. Az utolsó karaktere után 0 byte kerül. Használhatók az Escape-szekvenciák. Karakterláncon belül idézőjel \" alakban írható. Ha több sorban fér ki, a sort \ -el kell zárni.

A karakterláncot a fordítóprogram olyan static tárolási osztályú karaktertömbként kezeli, amelyet a megadott karakterek inicializálnak.

5.3.5 Operátorok

A C nyelvben nagyon sokféle operátor van, egy- két- és háromoperandusúak lehetnek. Az operátorokat teljes részletességgel a kifejezéseknél ismertetjük.

5.3.6 Egyéb elválasztók

Legfontosabb a `;` (pontosvessző), amely az utasítás végét jelzi, de ilyen a `{` és `}` is.

5.3.7 Megjegyzések

`/*` -al kezdődő, `*/` -el végződő karaktersorozat. Nem skatulyázható egymásba, de tetszőleges hosszúságú lehet.

6. A C nyelv alaptípusai

A C nyelvben előre definiált alaptípusokat, és ezekből származtatott összetett típusokat (aggregátumokat) használhatunk.

Ebben a pontban felsoroljuk az összes előre definiált típust és a helyfoglalásukat.

6.1 Integrális típusok

Típusnév	Hossz
`char`	1 byte
`int`	gépfüggő, a "természetes" hossz (szóhossz)
`short (int)`	(legalább) 16 bit
`long (int)`	(legalább) 32 bit

A nyelv csak azt írja elő, hogy a típusok hosszára a

```
short <= int és  int <= long
```

teljesüljön.

Mindegyik típusnév elé írható egy kulcsszó, ami az előjeles/előjel nélküli ábrázolást írja elő, az alábbiak szerint:

- `signed` (előjeles) - a char kivételével ez a default
- `unsigned` (előjel nélküli)

A `char` típus előjelessége gépfüggő. Csak az biztos, hogy minden nyomtatható karakter pozitív, mint egész szám.

A `char` típus a neve szerint karakterek kezelésére szolgál. Mivel azonban egy karakter tárolása a kódjának, mint egész számnak a tárolását jelenti, ez a típus minden további nélkül használható rövid egész számként, és alkalmazható minden aritmetikai kifejezésben. Valójában a C nyelv tehát nem tartalmaz karakter típust.

6.2 Lebegőpontos típusok

Típusnév	Hossz
`float`	gépfüggő (általában 4 byte)
`double`	gépfüggő (általában 8 byte)
`long double`	gépfüggő (általában 16 byte)

6.3. A `void` típusnév

A `void` típusnevet a C nyelv speciális célokra tartja fenn.

7. Kifejezések és operátorok

7.1 A balérték fogalma

Objektumnak nevezzük a C nyelvben a memória valamely műveletekkel kezelhető részét.

A balérték objektumra hivatkozó kifejezés. (Tulajdonképpen amelynek meghatározott címe van a memóriában.) Kézenfekvő példa a változó, de - mint később látni fogjuk - bonyolultabb szerkezet is lehet. Nevét onnan kapta, hogy állhat értékadás baloldalán. (De nem csak ott!)

7.2 Kifejezések és kiértékelésük

A kifejezés operandusok és operátorok sorozata. A kiértékelés sorrendjét az operátorok precedenciája határozza meg, és az, hogy balra vagy jobbra kötnek.

A precendencia - sorrendtől eltérő kiértékelési sorrendet zárójelezéssel írhatunk elő.

Kommutatív és asszociatív operátorokat (*, +, &, |, ~) tartalmazó kifejezések kiértékelési sorrendje (még zárójelezés esetén is!) meghatározatlan.

A kiértékelés során konverziók történ(het)nek: ezt nevezzük szokásos aritmetikai konverziónak. A konverzió pontos szabályait a nyelv definíciója rögzíti (lásd pl. [2] 215. oldal), de itt nem idézzük, mert első olvasásra bonyolultnak tűnik. A lényege az, hogy mindig a pontosabb számolás és az adatvesztés elkerülésének irányába konvertálódnak a típusok. Így például az egész lebegőpontossá, a rövid hosszabbá, az előjeles előjel nélkülivé alakul, ha szükséges.

7.3 Operátorok

Az operátorok felsorolása a precedenciájuk *csökkenő* sorrendjében történik. Egy alponton belül a precedencia azonos és a felsorolás teljes.

7.3.1 Elsődleges operátorok

( ) : zárójelek
[ ] : indexelés
. : hivatkozás struktúra-tagra
-> : hivatkozás struktúra-tagra struktúra-mutatóval

Csoportosítás balról jobbra.

7.3.2 Egyoperandusú operátorok

Csoportosítás jobbról balra.

`*kifejezés`	indirekció (hivatkozás adott címen levő értékre)
`&balérték`	mutató - képzés
`+kifejezés`	egyoperandusú + (szimmetria okokból)
`-kifejezés`	egyoperandusú -
`!kifejezés`	logikai nem: ha a kifejezés 0, az eredmény 1, ha a kifejezés nem 0, az eredmény 0. Egész értéket ad.
`~kifejezés`	az egész kifejezés értékének 1-es komplemense. (Bitenkénti negációja)
`++ balérték`	inkrementálás
`-- balérték`	dekrementálás
`(típusnév)kifejezés`	a kifejezés a megadott típusúvá alakul át ("cast")
`sizeof kifejezés`	az operandus mérete byte-ban
`sizeof (típusnév)`	az adott típus mérete byte-ban

A `balérték ++` és a `balérték--` speciális C nyelvi operátorok. Hatására a balérték értéke eggyel nő vagy csökken. Ha a ++ vagy -- a balértéktől balra van, az érték megváltozik, és ezután ez a megváltozott érték kerül felhasználásra. Ha a ++ vagy -- a balérték után helyezkedett el, a balérték felhasználódik, majd utána változik meg az értéke.

7.3.3 Multiplikatív operátorok

Csoportosítás balról jobbra.

`*` : szorzás
`/` : osztás
`%` : maradékképzés

Pozitív egészek osztása esetén az eredmény csonkul, ha valamelyik negatív, az eredmény gépfüggő lehet. (Általában az osztandó és a maradék előjele megegyezik.)

Az `a%b` az a- nak b- vel való osztása során kapott maradékot jelenti. Az a-nak és b-nek itegrális típusúnak kell lennie.

Mindig igaz, hogy

`(a/b)*b + a%b = a` (ha b # 0)

7.3.4 Additív operátorok

Csoportosítás balról jobbra.

+ : összeadás
- : kivonás

7.3.5 Léptető operátorok

Csoportosítás balról jobbra.

`a<<b` az a-t, mint bitmintát balra lépteti b bittel, a jobboldalon 0 bitek lépnek be.

`a>>b` mint fennt, de jobbra léptet. A belépő bit 0, ha `a` típusa `unsigned`, egyébként az előjel bit lép be.

A művelet végzése előtt az előzőekben említett aritmetikai konverziók végrehajtódnak. Az eredmény `int` típusú lesz. `a` és `b` csak egész típusú lehet, `b`-nek pozitívnak kell lennie. Ha `b` negatív, vagy értéke túl nagy, az eredmény határozatlan.

7.3.6 Relációs operátorok

Csoportosítás balról jobbra.

`<` `>` `<=` `>=`

Értékük `int` típusú, és 0 (hamis) vagy 1 (igaz). Megjegyzés: mutatók is összehasonlíthatók!

7.3.7 Egyenlőségi operátorok

Csoportosítás balról jobbra.

== : egyenlô
`!` = : nem egyenlő

Értékük `int` típusú, és 0 (hamis) vagy 1 (igaz). Megjegyzés: mutatók is összehasonlíthatók!

7.3.8 Bitenkénti ÉS operátor

Jele: `&` Mindkét operandusnak integrális típusúnak kell lennie.

7.3.9 Bitenkénti kizáró VAGY

Jele: `^` Mindkét operandusnak integrális típusúnak kell lennie.

7.3.10 Bitenkénti megengedő VAGY

Jele: | Mindkét operandusnak integrális típusúnak kell lennie.

7.3.11 Logikai ÉS

Jele: `&&` Mindkét operandusnak valamilyen alaptípusnak vagy mutatónak kell lennie. Az eredmény 0 (hamis) vagy 1 (igaz). Balról jobbra hajtódik végre, és *a második operandus nem értékelődik ki, ha az első értéke 0!*

7.3.12 Logikai VAGY

Jele: `||` Mindkét operandusnak valamelyik alaptípusnak vagy mutatónak kell lennie. Az eredmény 0 (hamis) vagy 1 (igaz). Balról jobbra értékelődik ki, és *a második operandus nem értékelődik ki, ha az első értéke nem nulla.*

7.3.13 Feltételes operátor

A feltételes kifejezés formája:

```
k1 ? k2: k3
```

Balról jobbra csoportosít. Végrehajtása: kiértékelődik az első kifejezés, és ha annak értéke nem 0, az eredmény a `k2` lesz, egyébként a `k3`. A `k2` és a `k3` közül csak az egyik (a szükséges) értékelődik ki.
Például az

```
a = k1? k2 : k3
```

értékadás egyenértékű az

```
if (k1)
  a=k2;
else
  a=k3;
```

programrészlettel.

7.3.14 Értékadó operátorok

Mindegyik értékadó operátor jobbról balra csoportosít. Két fajtája van:

1. egyszerű értékadó operátor
   Formája:

   ```
   balérték = kifejezés
   ```

A `kifejezés` kiértékelődik, a `balérték` (esetleg konverzió után) felveszi ezt az értéket, és ez lesz a művelet értéke is.

2. összetett értékadó operátor.
   Formája:

   ```
   balérték x= kifejezés
   ```

ahol `x` az alábbi műveletek egyike lehet:

+, -, *, /, %, >>, <<, &, ^, !

Az

```
E1 x= E2
```

(E1, E2 kifejezések) hatása ugyanaz, mint az

```
E1=E1 x E2
```

kifejezésnek, de az E1 csak egyszer értékelődik ki.

Megjegyzés

A C nyelvben az értékadó kifejezés *kétarcú*: pontosvesszővel lezárva értékadó utasításként viselkedik, de írható bárhová, ahová kifejezést lehet írni, és ilyenkor értéke a baloldal értéke.

Példa: az alábbi két programrészlet egyenértékű:

```
a=kifejezés
  if(a>10) utasítás;

if ( (a=kifejezés)>10 ) utasítás;
```

Itt az `a=kifejezés` körüli zárójel nem fölösleges, mert az = operátor precedenciája kisebb, mint a > operátoré. Az `a=kifejezés > 10` egyenértékű az `a=(kifejezés>10)` alakkal, aminek hatására `a` a 0 vagy 1 értéket veszi föl.

7.3.15. Kifejezés lista

Formája:

```
K1,K2
```

ahol `K1` és `K2` kifejezések. Hatására előbb a `K1`, majd a `K2` kifejezés kiértékelődik. A művelet eredménye a `K2` értéke.

7.3.16. Az operátor jelek összefoglaló táblázata

A könnyebb áttekinthetőség kedvéért táblázatba foglalva megismételjük az operátor jeleket.

Csoport	Operátorok	Asszociativitás
elsődleges	`(), [], ->, .`	b-j
egyoperandusú	`cast, sizeof, &, *, ++, --, ~ !`	j-b
multiplikatív	`*, /, %`	b-j
additív	`+, -`	b-j
eltolás	`<<, >>`	b-j
relációs	`<, <=, >, >=`	b-j
egyenlőség	`==, !=`	b-j
AND	`&`	b-j
XOR	`^`	b-j
OR	`|`	b-j
logikai AND	`&&`	b-j
logikai OR	`||`	b-j
értékadás	`=, +=, -=, /=, %=, >>=, <<=, &=, ^=`	j-b
kif. lista	`, (vessző)`	b-j

8. Állandó kifejezések

Állandó kifejezés az, amelyben

- egész állandók
- karakterállandók
- `sizeof` kifejezések

valamint az alábbi operátorok szerepelhetnek:

```
+ - * / % & I ^ << >> == !=  < > <= >=
```

Zárójelezés és feltételes kifejezés a fenti elemekből megengedett. Valós konstans vagy változó kezdőértékadásához valós operandusok is lehetnek.

Használhatók:

- kezdeti értékként
- tömbdeklarációban
- case szerkezetben

9 Tömbök

A tömb lehetővé teszi, hogy egy név alatt összefoglalhassunk több értéket. A tömb valamennyi eleme azonos típusú. Az egyes értékekre indexeléssel (vagy mutatókifejezéssel) hivatkozhatunk. Az indexek számától függően a tömb lehet egy- vagy többdimenziós.

9.1 Tömbdeklaráció

A tömböket deklarálni kell. A deklarációban meg kell adni a tömb nevét, indexeinek számát és az elemek darabszámát. Az indexek alsó határa 0!

A deklaráció formája:

```
tipus név[dbszám]{[dbszám]...}
```

ahol "dbszám" az elemek száma, tetszőleges állandó kifejezés. Példák:

```
char line[80];
int matrix[50][60];
```

A tömbelemek száma inicializálással közvetetten is megadható! (Részletesen lásd később!)

9.2 Tömbelem - hivatkozás

A tömb egyes elemeire való hivatkozásnál minden index helyére egy egész kifejezés írható. A kifejezés deklarált határok közé esését általában nem ellenőrzi a program!

10. Utasítások

Az alábbiakban felsoroljuk a C nyelv valamennyi utasítását, szintaktikai és szemantikai leírásával. Az utasítások formájának leírásánál a kulcszavakat és az utasítás egyéb kötelező "tartozékait" vastagított szedéssel emeljük ki.

10.1 Kifejezés utasítás

Formája:

```
kifejezés;
```

A kifejezés legtöbbször értékadás vagy függvényhívás.

10.2 Összetett utasítás vagy blokk

Formája:
összetett utasítás:

```
{
utasítások
}
```

blokk:

```
{
deklarációk
utasítások
}
```

Összetett utasítás mindenütt lehet, ahol a szintaktikában "utasítás" szerepel. Lehetséges (de **kerülendő**) az összetett utasítás belsejébe való ugrás.

10.3 A feltételes utasítás

Formái:
1.

```
if (kifejezés ) utasítás1;
```

2.

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

Kiértékelődik a `kifejezés`. Ha értéke nem nulla, `utasítás1` hajtódik végre. Ha a kifejezés 0, akkor a 2. formánál az `utasítás2` hajtódik végre, majd mindkét formánál a következő utasítással folytatódik a program.

`utasítás1` és `utasítás2` újabb feltételes utasításokat tartalmazhat. Az egymásba skatulyázás szabálya: egy `else` mindig az utoljára talált `else` nélküli `if`-hez kapcsolódik!

Speciális esete az alábbi

```
if (kif1)
  ut1
else if (kif2)
  ut2
else if (kif3)
  ut3
    .
    .
    .
else
  utn;
```

feltétellánc, amelyben a feltételek sorban értékelődnek ki az első nem nulla eredményig. Csak az ehhez a feltételhez tartozó utasítás hajtódik végre.

10.4 A `while` utasítás

Formája:

```
while ( kifejezés ) utasítás;
```

Az `utasítás` mindaddig ismétlődik, míg `kifejezés` értéke nem nulla. A kifejezés kiértékelése az utasítás végrehajtása előtt történik.

10.5 A `do` utasítás

Formája:

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

Hasonlóan működik, mint a `while`, de a vizsgálat az utasítás végrehajtása után történik.

10.6 A `for` utasítás

Formája:

```
for ( k1; k2; k3) utasítás;
```

Egyenértékű az alábbi programrészlettel:

```
k1;
while (k2)
  {
  utasítás;
  k3;
  }
```

Ha `k2` hiányzik, helyére `1` íródik. Ha `k1` vagy `k3` elmarad, a fenti kifejtésből is elmarad.

10.7 A `switch` utasítás

Formája:

```
switch (kif)
  {
  case (ak1):
    u1;
  case (ak2):
    u2;
  case (akn):
    un;
  default:                    <-- ez nem kötelező!
    un+1;
  };
```

ahol `kif` egy `int` értéket adó kifejezés kell legyen, `ak1, ak2` stb. állandó kifejezések, `u1, u2` stb. pedig utasítások.

Működés:

1. kiértékelődik a `kif`
2. a program azon első `case` szerkezet utáni utasítással folytatódik, amelyben szereplő állandó kifejezés értéke egyenlő `kif` értékével.
3. ha a fenti feltétel egy esetben sem teljesül, a `default` utáni utasítással folytatódik (ha van ilyen címke).
4. minden egyéb esetben a `switch` utasítást követő utasítással folytatódik a program.

10.8 A `break` utasítás

Formája:

```
break;
```

Hatására befejeződik a `break`-et körülvevő legbelső `while, for, do` vagy `switch` utasítás végrehajtása, és a vezérlés átadódik a következő utasításra.

10.9 A `continue` utasítás

Formája:

```
continue;
```

Hatására a vezérlés a körülvevő legbelső `while, do` vagy `for` utasítás ciklusvégére adódik át.

10.10 A `return` utasítás

Formája:

```
return;
```

A függvény végrehajtása befejeződik, és a hívó függvényhez tér vissza a vezérlés. A függvény értéke definiálatlan.

```
return kifejezés;
```

Visszatérés a hívó függvényhez, a függvény értéke a `kifejezés` lesz. (Típuskonverzióval, ha szükséges.)

10.11 A `goto` utasítás

Formája:

```
goto azonosító;
```

A vezérlés az `azonosító` címkéjű utasításra kerül.

10.12 A címkézett utasítás

Bármelyik utasítást megelőzheti az

```
azonosító:
```

alakú címke, és így `goto` utasítás célpontja lehet.

10.13 Az üres utasítás

Egy magában álló pontosvessző. Legtöbbször azért használjuk, mert címkéje lehet, vagy mert a ciklustörzs üres.

11. Egyszerű input-output

A C nyelv nem tartalmaz input-output utasításokat, ezeket szabványos függvényekkel oldja meg. Az úgynevezett standard input (ami alapesetben a billentyűzet) és a standard output (alapesetben a képernyő) kezelésére szolgáló legegyszerűbb függvényeket ismertetjük itt, hogy a példaprogramokban az adatkezelést meg tudjuk oldani.

Az alábbi függvényeket használó forrásfile elején a

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

sornak (aminek jelentését majd csak később tudjuk megmagyarázni) szerepelnie kell.

11.1. Karakter beolvasása a standard inputról

A standard inputról egy karaktert olvas be a

```
getchar()
```

függvény. Visszatérési értéke a beolvasott karakter kódja, vagy az `EOF` előre definiált állandó, ha elértük a file végét. Az `EOF` állandó értéke gépfüggő, és nem biztos, hogy "belefér" a `char` típusba.

11.2. Egy karakter kiírása a standard outputra

Erre szolgál a

```
putchar(char c)
```

függvény, amely a paramétereként megadott karaktert a standard outputra írja.

11.3. Formázott kiírás a standard outputra

A `printf` függvényt használhatjuk erre a célra.

Formája:

```
printf ("formátum-specifikáció",arg1,arg2, ...);
```

A függvény az `arg1, arg2, ...` argumentumok értékét az első paraméterének megfelelő módon konvertálja és kiírja a standard outputra.

A formátum-specifikáció tartalmazhat:

- közönséges karaktereket
  ezeket változtatás nélkül kinyomtatja. Escape szekvenciákat is tartalmazhat.
- konverzió-specifikációkat
  ezek a soron következő argumentum nyomtatási formátumát határozzák meg.

A legfontosabb formátum-specifikációk:

%nd	Egész érték kiírása n karakter széles helyre, balra igazítva
%d	Egész érték kiírása a szükséges szélességben
%s	Karakterlánc kiírás végig (a \0-t tartalmazó byte-ig)
%ns	Karakterlánc első n karakterének kiírása, ha kell, balra igazítva
%n.mf	Lebegőpontos szám fixpontos kiírása n szélességben, m tizedesjeggyel
%n.me	Lebegőpontos szám kiírása lebegőpontos formában, n szélességben, a karakterisztikában m tizedesjegyet használva

A függvény az argumentumok számát az első paraméteréből határozza meg. Ha ez az általunk megadott paraméterek számával nem egyezik meg, a program viselkedési kiszámíthatatlan! Hasonló problémát okozhat egy karaktertömb kiírása a `%s` formátummal, ha nem gondoskodtunk a záró 0 byte-ról.

Egyszerű példák:

```
printf ("Egy egy szöveg\n");

int a;
int i;
float b;
char c,ch[5];
a=1;
b=2;
c='A';
for (i=0; i<5; i++) ch[i] = 'a' + i;
ch[5] = '\0';
printf ("a=%3d b=%5.1f c=%c ch=%s\n",a,b,c,ch);
```

12. Az első példaprogramok

A C nyelv eddig megismert elemei segítségével írjunk meg néhány egyszerű programot.

12.1. Példaprogram

Írjunk programot, amely kiírja a kisbetűk kódjait!

```
/*  CPELDA1.C  */
/* Készítette: Ficsor Lajos */

/************************************************
  A program kiírja a kisbetűk kódjait.
************************************************/

#include <stdio.h>

void main(void)
{
```

```
char i;

/* Fejléc írása */
printf ("\n A kisbetuk kodjai:\n");

/* A ciklus végigmegy a kisbetűkön */
for (i='a'; i<='z'; i++)
  {
  printf ("Betu: %c  Kodja: %d\n", i, i);
  }
}
```

A fenti kis program megmutatja, hogy ugyanazon változó különböző konverziókkal is kiírható. Egyben szemlélteti, hogy a `char` típus valójában egészként kezelhető.

Fontos megjegyezni, hogy a program megírásához nem kellett ismerni a kódtáblát, csak azt kellett feltételezni róla, hogy a kisbetűk folyamatosan, egymás után helyezkednek el benne. Ez a legáltalánosabban használt ASCII kódtáblára és az ékezet néküli betűkre igaz.

12.2 Példaprogram

Írjunk programot, amely a standard bemenetről beolvasott szöveget csupa nagybetűvel írja ki.

```
/*  CPELDA2.C  */
/* Keszitette: Ficsor Lajos */

/*****************************************************
  A program a standard bemenetről beolvasott szöveget
  nagybetűsen írja ki
*****************************************************/

#include <stdio.h>

void main(void)
{
int c; /* int típusú, hogy az EOF is ábrázolható legyen! */

while ( (c=getchar()) != EOF)  /* Olvasás file végéig */
  {
  if (c >= 'a' && c<= 'z')     /* Ha kisbetűt olvastunk be */
    {
    c = c - ('a' - 'A');       /* Konvertálás nagybetűre */
    }
  putchar(c);                  /* Karakter kiírása */
  }
```

A fenti program azt tételezi fel, hogy a kódtábla mind a kisbetűket, mind a nagybetűket folyamatosan tárolja, és az azonos kisbetű és nagybetű közötti "távolság" (a kódjaik különbsége) állandó. Ez az ASCII kódtáblára és az ékezet nélküli betűkre igaz.

A

```
while ( (c=getchar()) != EOF)
  {
  utasítások
  }
```

szerkezet egy file karakterenkénti beolvasását és feldolgozását végző szokásos megoldás.

13. A függvény

A függvény (mint minden programozási nyelvben) utasítások egy csoportja, amelyek megadott paramétereken képesek műveleteteket végezni. A tipikus C nyelvű program sok, viszonylag egyszerű függvény összessége.

13.1. Függvény definíció

Formája:

```
típus név (formális paraméterlista)
  {
  lokális deklarációk
  utasítások
  }
```

A formális paraméterlista

```
típus azonosító
```

vagy

```
típus tömbnév[]
```

párok, vesszővel elválasztva. Ha nincs paramétere, a paraméterlista helyére a `void` alapszó írandó.

A visszatérési érték típusa bármely típusnév lehet. Ha a függvény nem ad vissza értéket, a visszatérési érték típusa `void`.

13.2. Függvény deklaráció (prototípus)

A függvényt a használata előtt deklarálni kell. A függvény deklaráció a függvény definíció fejével egyezik, és pontosvessző zárja. Formája tehát:

```
típus név (formális paraméterlista);
```

Megjegyzés

A C nyelv a fentiektől enyhébb szabályokat ír elő, de a helyes programozási stílus elsajátítása érdekében fogadjuk el ezt a szigorúbb szabályozást.

13.3. A függvény hívása:

```
név (aktuális paraméterlista)
```

A függvényhívás állhat magában, pontosvesszővel lezárva, (ekkor a visszaadott érték - ha volt - elvész), vagy kifejezés részeként. Az aktuális paraméterlista kifejezések vesszővel elválasztott listája. A zárójel pár kiírása akkor is kötelező, ha nincs paraméterlista!

A C nyelv csak az *érték szerinti paraméterátadási mechanizmust* ismeri. Ez a következő folyamatot jelenti:
1. kiértékelődik az aktuális paraméter kifejezés
2. a kifejezés értéke a formális paraméter típusára konvertálódik a szokásos típuskonverzió szabályai szerint
3. a formális paraméter megkapja kezdőértéknek ezt az értéket
4. végrehajtódnak a függvény törzsében felsorolt utasítások.

A fenti szabályok értelmében a formális paraméterek a függvényre nézve *lokális* változóknak tekinthetők (a fogalom pontos magyarázatát csak később tudjuk megadni), amelyek az aktuális paraméter kifejezés értékével inicializálódnak. A formális paraméterek a függvényen belül kaphatnak más értéket is, de ennek az aktuális paraméterre semmi hatása nincs.

13.4. Példaprogram: faktoriális számítása

Bevezető példaként írjunk egy függvényt, amely a faktoriális értékét számítja ki. Írjunk egy teljes programot, amely ezt a függvényt használja.

```
/*  CPELDA3.C  */
/* Készítette: Ficsor Lajos */
/*****************************************************
  Függvény n! számításához.
  Próbaprogram a függvényhez.
*****************************************************/

#include <stdio.h>
long faktor (int n);  /* Ez a függvény deklarációja */

void main(void)       /* Főprogram */
{
int n;

n= 10;
/* az alábbi sor tartalmazza a fgv hívását */
printf ("\n%d faktorialisa: %ld",n, faktor(n));
}

long faktor (int n)   /* Függvény definíció fejrésze */
/*****************************************************
  A függvény n! értéket számítja.
  Nem rekurzív.
*****************************************************/
{
long fakt;
int i;                /* lokális deklarációk */
```

```
for (fakt=1, i=2; i<=n; i++) fakt *= i;

return fakt;
}
```

Érdemes megnézni a ciklusutasítás megoldását:

`fakt=1, i=2;`	Kezdőérték beállítása. A vessző operátor teszi lehetővé egynél több kifejezés írását.
`i<=n;`	A ciklus leállításának a feltétele.
`i++;`	Léptetés. Ebben az esetben a `++i` is ugyanazt a hatást érte volna el.
`fakt *= i;`	A ciklus magja. Összetett értékadó operátort használ a **`fakt = fakt*i`** kifejezés egyszerűsítésére.

Megjegyezzük még, hogy ezt a ciklusutasítást a gyakorlottabb C programozó az alábbi tömörebb formában írta volna fel:

```
for (fakt=1, i=2; i<=n; fakt *= i++);
```

Ebben az alakban kihasználtuk a léptető operátor azon tulajdonságát, hogy előbb az i értéke felhasználódik a kifejezés kiszámításánál, majd *utána* növelődik eggyel az értéke. A ciklusváltozó léptetése tehát most egy műveletvégző utasítás mellékhatásaként történik meg. Ez az egyetlen kifejezés tehát egyenértékű az alábbi utasításokkal:

```
fakt = fakt *i;
i = i+1;
```

Ezzel a megoldással minden feladatot a ciklusutasítás vezérlő részére bíztunk, így a ciklustörzs üres: ezt jelzi az utasítás után közvetlenül írt pontosvessző.

13.5. Példaprogram: egy sor beolvasása

A függvényírás gyakorlására írjunk egy következő függvényt, amely beolvas egy sort a standard inputról, és azt egy stringként adja vissza. Ehhez a következőket érdemes végiggondolni:

- Egy sor végét a `\n` (sorvég) karakter jelzi.
- A string (karaktersorozat) tárolására a `char` típusú tömb a legalkalmasabb.
- Érdemes betartani azt a C nyelvi konvenciót, hogy a string végét egy 0 tartalmú byte jelzi, mert ekkor azt a szokásos módon kezelhetjük (pl `%s` konverzióval kiírathatjuk, használhatjuk rá a szabványos string kezelő függvényeket).

Mindezek figyelembevételével a függvény és azt azt használó főprogram például az alábbi lehet:

```
/*  CPELDA4.C  */
/* Készítette: Ficsor Lajos */

/*****************************************************
  Egy sor beolvasása függvénnyel.
  Próbaprogram a függvényhez.
*****************************************************/

#include <stdio.h>

#define MAX 100

int getstr (char s[]);

void main(void)
{
int n;
char szoveg[MAX+1];

printf ("\nEgy sor beolvasasa a standard inputrol\n\n");

n = getstr(szoveg);
printf ("%s\n",szoveg);
printf ("A szoveg hossza: %d\n",n);

}

int getstr (char s[])
/*****************************************************
  A függvény egy sort olvas be a standard inputról,
  es az s tömbbe helyezi el, string-ként.
  Visszatérési értéke a beolvasott karakterek száma,
  vagy EOF
*****************************************************/
{
int c;
int i;

i=0;
while ( (c=getchar()) != '\n' && c !=EOF)
  {
  s[i++] = c;
  }
s[i] = '\0';/* Záró 0 elhelyezése */
return c==EOF ? c : i;
}
```

Megjegyzések a programszöveghez:

1. A
   `#define MAX 100`
   sor *szimbolikus konstanst* definiál. A jó programozási stílushoz tartozik, hogy a konstansoknak

olyan nevet adjunk, amely a jelentésére utal. Így olvashatóbbá tesszük a szöveget, és egy esetleges változtatáshoz csak egyetlen helyen kell módosítani a programot. Az itt alkalmazott szerkezet úgynevezett *makró definíció*. Ezt egy előfeldolgozó program (precompiler) a tényleges fordítóprogram előtt feldolgozza úgy, hogy a programszöveget végignézve minden MAX karaktersorozatot az 100 karaktersorozattal helyettesít. A fordítóprogram tehát a
`char szoveg[100+1];`
sort kapja meg.

2. A `szoveg` karaktertömb definíciója azt jelenti, hogy maximum 100 karakter hosszúságú string tárolására alkalmas, ekkor a string végét jelző 0 byte a 101. elembe kerül. Ne felejtsük azonban el, hogy a C az indexelést 0-tól kezdi. A 101 elemű tömb legális indexei tehát 0-tól 100-ig terjednek, így az első karakter a "nulladik" tömbelem. Gyakori hibaforrás C programokban ennek a figyelmen kívül hagyása.

3. Mivel a beolvasó függvény nem figyeli a beolvasott karakterek számát, a fenti kis program hibásan működik, ha 100-nál több karakterből álló sort kap. Ilyenkor a tömb nem létező elemeibe ír, amely előre nem meghatározható (és akár futásonként más és más) hibajelenséget idézhet elő!

4. Az utolsó sor lehet, hogy nem sorvégjellel végződik, hanem a file vége (`EOF`) jellel. Ebben az esetben a függvény nem jól dolgozik: `EOF`-et ad vissza a függvényértékben, bár a paramétere tartalmazza a beolvasott stringet.

5. A
`while ( (c=getchar()) != '\n' && c !=EOF)`
sor "magyar fordítása": olvasd be a következő karaktert, és mindaddig, amíg az nem sor vége, és nem file vége, ismételd az alábbi utasításokat!

6. A `return` utasításban alkalmazott feltételes kifejezéssel az alábbi programrészletet tudtuk helyettesíteni:
```
if (c==EOF)
  return EOF
else
  return i;
```

7. A ciklus a `for` utasítással is megfogalmazható:
`for( i=0; (c=gethar()) != '\n' && c !=EOF); s[i++] = c)`
Bár ez sokkal tömörebb, emiatt nehezebben is olvasható, ezért talán szerencsésebb az eredeti megoldás. Idegen programok olvasásánál azonban számíthatunk ilyen jellegű részletekre is.

14. Mutatók

A mutató (pointer) olyan változó, amely egy másik objektum címét tartalmazza. (Ezért belső ábrázolási módja erősen gépfüggő!) A mutató értéket kaphat az `&` operátorral, a mutató által megcímzett tárrész pedig a `*` operátorral. Így tehát a

```
px = &x;
```

utasítás, ha `px` mutató, ahhoz az `x` változó *címét* rendeli. Ha ezután az

```
y = *px;
```

utasítást írjuk, a két utasítás együttes hatása azonos az

```
y=x;
```

értékadással.

14.1 Mutatók deklarációja

A mutatók deklarációjának tartalmaznia kell, hogy milyen típusú objektumra mutat. Formálisan:

```
típus *azonosító;
```

Például:

```
int *px;
```

A `*mutató` konstrukció **balérték**, tehát szerepelhet értékadó utasítás baloldalán. Például a `*px=0` a `px` által megcímzett egész értéket 0-ra állítja be, a `(*px)++` pedig inkrementáltja.

Megjegyzés

A `*px++` nem azonos a fentivel mivel az egyoperandusú operátorok jobbról balra csoportosítanak, így ennek zárójelezése a `*(px++)` lenne, ami azt jelenti, hogy a `px` inkrementálódik (ennek pontos jelentését lásd a következő alpontban), majd az így keletkezett címen levő értékre hivatkozunk.

14.2 Címaritmetika

Mivel a mutató is változó, így értéket kaphat, és műveletek végezhetők rajta. A műveletek definíciója figyelembe veszi azt a tényt, hogy a mutató címet tárol, az eredményt pedig befolyásolja az, hogy a mutató milyen típusra mutat.

Az alábbi műveletek megengedettek:

- •mutató és egész összeadása, kivonása
- mutató inkrementálása, dekrementálása
- két mutató kivonása
- mutatók összehasonlítása
- mutatónak "0" érték adása
- mutató összehasonlítása 0-val
- mutató indexelése
- mutatók közötti értékadás

A felsorolt műveleteken kívül minden más művelet tilos.

Az egyes műveletek definíciója:

- A mutató és egész közötti művelet eredménye újabb cím, amely ugyanolyan típusú objektumra mutat. Például

  ```
  tipus *p
  int n;
  ```

  esetén a `p+n` egy olyan cím, amelyet úgy kapunk, hogy a `p` értékéhez hozzáadunk egy

  ```
  n * sizeof (típus)
  ```

  mértékű eltolást. Ezáltal az eredmény az adott gép címzési rendszerében egy újabb cím, amely ugyanolyan típusú, de `n` elemmel odébb elhelyezkedő objektumra (például egy tömb `n`-el nagyobb indexű elemére) mutat. Így például, ha

  ```
  int *px,n
  ```

  akkor a

  ```
  px+n
  ```

  kifejezés eredménye mutató, amely a `px` által megcímzett egész utáni `n.` egészre mutat. Hasonlóan értelmezhetők a `p-n p++ p--` kifejezések is.

- Két azonos típusú mutató kivonása mindig engedélyezett, de általában csak azonos tömb elemeire mutató pointerek esetén van értelme. Eredménye `int` típusú, és a két cím között elhelyezkedő, adott típusú elemek számát adja meg.

- Az azonos típusú `p1` és `p2` mutatókra a `p1 < p2` reláció akkor igaz, ha `p1` kisebb címre mutat (az adott gép címzési rendszerében), mint `p2`. Ez abban az esetben, ha mindkét mutató ugyanazon tömb elemeit címzi meg, azt mutatja, hogy a `p1` által címzett elem sorszáma kisebb, mint a `p2` által címzetté. Más esetben általában az eredmény gépfüggő, és nem értelmezhető. A többi reláció értelemszerűen hasonlóan működik.

- A 0 értékű mutató speciális jelentésű: nem mutat semmilyen objektumra. Ezzel lehet jelezni, hogy a mutató még beállítatlan. Ezért engedélyezett a 0-val való összehasonlítás is.

- A mutató indexeléséről bővebben a mutatók és tömbök összefüggésének tárgyalásánál beszélünk.

14.3 Mutatók és tömbök

A C- ben a tömbök elemei indexeléssel és mutatókkal egyaránt elérhetők. Ennek alapja az, hogy egy tömb azonosítóját a fordító mindig a tömb első elemét megcímző mutatóként kezeli. Ennek következményeit szemlélteti a következő összeállítás:

Legyen a deklarációs részben az alábbi sor:

```
int *pa,a[10],i;
```

és tételezzük fel, hogy végrehajtódott a

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

utasítás.

Ekkor értelmesek az alábbi kifejezések, és a megadott jelentéssel rendelkeznek.

Kifejezés	Vele egyenértékű	Jelentés
`pa=&a[0]`	`pa = a`	A `pa` mutató az `a` tömb első elemére mutat

`a[i]`	`*(pa+i)`
	`*(a+i)`
	`pa[i]`
		Hivatkozás az `a` tömb `i` indexű elemére

`&a[i]`	`pa+i` `a+i`	Az `a` tömb `i` indexű elemének címe

Megjegyzés

Bár a tömb azonosítója a fordítóprogram számára mutatóként viselkedik, mégsem változó, ebből következik, hogy nem balérték, így az

```
a=pa     pa++       p=&a
```

jellegű kifejezések tilosak!

14.4 Karakterláncok és mutatók

A fordítóprogram a karakterlánc-állandót (stringet) karakter típusú tömbként kezeli, és megengedi, hogy egy karaktertömböt karakterlánccal inicializáljunk. A karakterlánc végére a záró 0 byte is odakerül. Ugyanezen okból megengedett egy `char*` mutató és egy string közötti értékadás, hiszen ez a fordító számára két mutató közötti értékadást jelent. Például:

```
char *string;
string="Ez egy szoveg"
```

használható, és ez után `*string` egyenlő `'E'` -vel, `*(string+3)` vagy `string[3]` egyenlő `'e'`-vel `*(string+14)` egyenlő `'\0'`-val, `*(string+20)` pedig határozatlan.

14.5 Mutató-tömbök, mutatókat címző mutatók

Mivel a mutató is változó, így

- tömbökbe foglalható,
- címezheti mutató.

A

```
char* sor[100];
```

deklaráció egy 100 darab `char` típusú mutató tárolására alkalmas tömböt definiál. A `sor[1]` például a sorban a második mutató.

Mivel a tömb azonosítója is mutató, így a `sor` is az, de egy mutatót (`a sor[0]`-át) címez meg, azaz típusa `char**`. Így `sor[1]` azonos a `*(sor+1)`- el, és ha `px char*` típusú mutató, akkor a

```
px = sor[1]
px = *(sor+1)
```

kifejezések értelmesek.

14.6 Többdimenziós tömbök és mutatók

A kétdimenziós tömb úgy fogható fel, mint egy egydimenziós tömb, amelynek minden eleme tömb. (Ennek következménye az, hogy a C programban használt tömbök elemei sorfolytonosan tárolódnak a memóriában.)

A tömbazonosító pedig mutató, így például

```
int a[10][10]
int* b[10]
```

esetén `a[5][5]` és `b[5][5]` egyaránt írható, mindkét kifejezés egy-egy egész értéket ad vissza. Azonban az `a[10][10]` 100 egész szám tárolására szükséges helyet foglal el, `a b[10]` csak 10 cím számára szükségeset, és a `b[5][5]` felhasználása csak akkor értelmes, ha a `b[5]` elemeit előzőleg valahogyan beállítottuk.

A fentiek illusztrálására nézzünk egy példát. Legyen a deklarációban

```
char s[5][10];
char *st[5];
```

Ekkor írható

```
s[0]= "Első";
s[1]= "Második";
s[2]= "Harmadik";
s[3]= "Negyedik";
```

```
s[4]= "Ötödik";
```

Ebben az esetben az `s` tömb 50 byte-nyi helyet foglal le, és minden string hossza korlátozott 10- re. Az `s[1][5]` a fenti utasítások után az i betűt jelenti.

Mivel a deklarációban az
```
char *st[5];
```
is szerepel, írható

```
st[0] = "Első";
st[1] = "Második";
st[2] = "Harmadik";
st[3] = "Negyedik";
st[4]="Ötödik"
```

Ebben az esetben az `st[5]` 5 címnek szükséges helyet foglal el, amihez még hozzáadódik a karaktersorozatok tárolására szükséges hely (de csak a feltétlenül szükséges), ráadásul tetszőleges hosszúságú stringek tárolhatók. Az `st[1][5]` a fenti utasítások után is az i betűt jelenti.

14.7. Mutató, mint függvény argumentum

Ha egy függvény argumentuma tömb, ebben az esetben - mint a legtöbb összefüggésben - a tömbazonosító mutatóként viselkedik, azaz a függvény a tömb első elemének címét kapja meg. Ez azt is jelenti, hogy a deklarációban mind a tömb-szerű, mind a mutató típusú deklaráció alkalmazható, és a deklaráció formájától függetlenül a tömbelemekre mutatókon keresztül vagy indexeléssel is hivatkozhatunk. Ennek megfelelően az alábbi formális paraméter-deklarációk egyenértékűek:

```
int a[]      int *a
int b[][5]        int (*b)[5]
```

Függvény argumentuma lehet egy változó mutatója. Így indirekt hivatkozással megváltoztatható a függvény argumentumának értéke. Példaként egy függvény, amely megcseréli két argumentumának értékét:

```
void csere (int* x,int* y)
{
int a;
a = *x;
*x = *y;
*y = a;
}
```

A függvény hívása:

```
main()
{
int a,b;
  .
```

```
  .
csere (&a,&b);
  .
  .
}
```

14.8 Függvényeket megcímző mutatók

Bár a függvény nem változó, de van címe a memóriában, így definiálható függvényt megcímző mutató. Ezáltal lehet függvény más függvény paramétere, sőt ilyen mutató függvényérték is lehet. (Az ennek megfelelő deklarációkra példákat a későbbiekben, a deklarációkkal kapcsolatban adunk.)

14.9. Példaprogram: string másolása

Az alábbi példaprogram egy string-másoló függvény (a standard strcpy függvény megfelelője) három lehetséges megoldását mutatja. A C nyelv lehetőségeit kihasználva egyre tömörebb programszöveget kaphatunk.

```
/*  CPELDA5.C  */
/* Készítette: Ficsor Lajos */

/*****************************************************
  Példa string másolás különböző változataira
*****************************************************/

#include <stdio.h>

void strmasol1 (char* cel, char* forras);
void strmasol2 (char* cel, char* forras);
void strmasol (char* cel, char* forras);

void main(void)
{
char szoveg[50];

printf ("\n String masolas\n\n");

strmasol1(szoveg,"Ficsor Lajos");
printf("Elso verzio: %s\n",szoveg);

strmasol2(szoveg,"Ficsor Lajos");
printf("Masodik verzio: %s\n",szoveg);

strmasol(szoveg,"Ficsor Lajos");
printf("Legtomorebb verzio: %s\n",szoveg);

}
```

```
    void strmasol1 (char* cel, char* forras)
    {
    /*******************************************
      Tömb jellegű hivatkozás, teljes feltétel
    *******************************************/

    int i;

    i=0;
    while ( (cel[i] = forras[i]) != '\0') i++;
    }


    void strmasol2 (char* cel, char* forras)
    {
    /*******************************************
      Tömb jellegű hivatkozás, kihasználva,
      hogy az értákadás érteke a forrás karakter
      kódja, ami nem nulla.  A másolás le-
      állításának feltétele a 0 kódú karakter
      átmásolása. (0 => "hamis" logikai
      érték!)
    *******************************************/

    int i;

    i=0;
    while ( cel[i] = forras[i] ) i++;
    /* A BORLAND C++ reakciója a fenti sorra:
       Warning: Possibly incorrect assigment
       Figyelem: lehet, hogy hibás értékadás.
       Magyarázat: felhívja a figyelmet arra
       a gyakori hibára, hogy = operatort írunk
       == helyett. Itt természetesen a sor
       hibátlan!
    */
    }

    void strmasol (char* cel, char* forras)
    {
    /*******************************************
      Pointer jellegű hivatkozas, kihasználva,
      hogy a paramértátadás érték szerinti,
      tehát a formális paraméterek segéd-
      változóként használhatók.
    *******************************************/
    while ( *cel++ = *forras++ );
    /* A BORLAND C++ reakciója a fenti sorra
       ugyanaz, mint az előző verziónál. A sor
       természetesen ebben a függvényben is
       hibátlan!
    */
}
```

14.10 Példaprogram: állapítsuk meg egy stringről , hogy numerikus-e

Készítsünk egy függvényt, amely a paraméterként kapott string-ről megállapítja, hogy csak helyköz és számjegy karaktereket tartalmaz-e. Írjunk **main** függvényt a kipróbáláshoz. A string beolvasására használjuk a CPELDA4.C-ben megírt *getstr* függvényt!

```
/*  CHF1.C  */
/* Készítette: Ficsor Lajos */

/*********************************************************
  A feladat olyan függveny írása, amely egy stringről
  megállapítja, hogy csak helyköz és számjegy karaktereket
  tartalmaz-e.
  Egy sor beolvasása a getstr függvénnyel történik.
  Próbaprogram a függvényhez.
*********************************************************/

#include <stdio.h>

#define MAX 100
#define IGAZ 1
#define HAMIS 0

int getstr (char s[]);
int szame1(char* s);
int szame(char* s);

void main(void)
{
int n;
char szoveg[MAX+1];

printf ("\nEgy sorol megallapitja, hogy csak\n");
printf ("helykozt es szamjegyet tartalmaz-e\n");

printf ("Elso verzio:\n");
getstr(szoveg);
if ( szame1(szoveg) )
  printf("\nNumerikus!\n");
else
  printf("\nNem numerikus!\n");

printf ("Masodik verzio:\n");
getstr(szoveg);
if ( szame(szoveg) )
  printf("\nNumerikus!\n");
else
  printf("\nNem numerikus!\n");
```

```
}


int getstr (char* s)
/*****************************************************
  A fuggveny egy sort olvas be a standard inputrol,
  es az s tombbe helyezi el, string-kent.
  Visszateresi erteke a beolvasott karakterek szama,
  vagy EOF
*****************************************************/
{
int c;
int i;

i=0;
while ( (c=getchar()) != '\n' && c !=EOF)
  {
  s[i++] = c;
  }
s[i] = '\0';
return c==EOF ? c : i;
}


int szame1(char* s)
/************************************************************
  A függvény IGAZ (1) értékkel tér vissza, ha a paraméter
  string csak helyköz vagy számjegy karaktereket tartalmaz,
  HAMIS (0) értekkel egyébként.
  Az üres sztringre IGAZ értéket ad.
  Tömb stílusú hivatkozásokat használ.
************************************************************/
{
int szamjegy_e;
int i;
char c;

szamjegy_e = IGAZ;
i =0;
while ( c=s[i++] )    /* Kihasználja, hogy a záró 0 leállítja*/
  {                   /* a ciklust. Az i++ kifejezés lépteti */
                      /* a ciklusváltozót.                   */
  if ( !(c==' ' || c>='0' && c<='9') )
    {
    szamjegy_e = HAMIS;
    break;
    }
  }
return szamjegy_e;
}
```

```
int szame(char* s)
/*************************************************************
  A függvény IGAZ (1) értékkel tér vissza, ha a paraméter
  string csak helyköz vagy számjegy karaktereket tartalmaz,
  HAMIS (0) értekkel egyébként.
  Az üres sztringre IGAZ értéket ad.
  Pointer stílusú hivatkozásokat használ.
  Nem használ break utasítást.
*************************************************************/
{
int szamjegy_e;
char c;

szamjegy_e = IGAZ;
while ( (c=*s++) && szamjegy_e)
  {
  if ( !(c==' ' || c>='0' && c<='9') ) szamjegy_e = HAMIS;
  }
return szamjegy_e;
}
```

14.11 Példaprogram: string konvertálása egész számmá

Írjunk függvényt, amely egy csak helyközt és számjegyeket tartalmazó string-ből kiolvas egy egész számot. (Szám határoló karakter a helyköz vagy a string vége). Visszatérési érték a szám, és paraméterben adja vissza a feldolgozott karakterek számát is.
A fenti függvény segítségével olvassuk be egy stringben levő valamennyi számot.

```
/*  CPELDA6.C  */
/* Készítette: Ficsor Lajos */

/*************************************************************
  Egy csak helyközt és számjegyeket tartalmazó stringből
  egész számokat olvas be. Szám határoló jel: helyköz.
*************************************************************/

#include <stdio.h>

int aboli (char* s, int* hossz);
int getstr (char* s);

void main(void)
{
char szoveg[100];
int sorhossz,kezdes,sorszam,szam,hossz;

sorhossz = getstr(szoveg);

kezdes = 0;
sorszam = 0;
```

```
while ( kezdes < sorhossz) /* Mindaddig, amíg a string */
                           /* végére nem érünk          */
  {
  /* A soron következő szám beolvasása */
  szam = aboli(szoveg+kezdes, &hossz);
  printf ("A(z) %d. szam: %d, hossza: %d\n", ++sorszam,
           szam,hossz);
  /* A következő szám kezdetének beállítása */
  kezdes += hossz;
  }
}

int aboli(char* s, int* hossz)
/*********************************************************
  Az s stringből egész számokat olvas be. Határoló jel:
  legalább egy helyköz vagy a string vége.. A string elején
  álló helyközöket átugorja.
  Visszatérési érték a beolvasott szám, a második paraméterben
  pedig a feldolgozott karakterek hossza. (típusa ezért int*!)
*********************************************************/
{
int i;
int szam;

i=0;
szam = 0;
/* Vezető helyközök átugrása /*
while ( s[i] == ' ') i++;
/* Szám összeállítása */
while ( s[i] != ' ' && s[i] != '\0' )
  {
  szam = 10*szam + s[i++] - '0';
  }
*hossz = i; /* Mutatón keresztüli indirekt hivatkozás */
return szam;
}

int getstr (char* s)
/*****************************************************
  A fuggveny egy sort olvas be a standard inputrol,
  es az s tombbe helyezi el, string-kent.
  Visszateresi erteke a beolvasott karakterek szama,
  vagy EOF
*****************************************************/
{
int c;
int i;

i=0;
while ( (c=getchar()) != '\n' && c !=EOF)
  {
  s[i++] = c;
  }
```

```
s[i] = '\0';
return c==EOF ? c : i;
}
```

Megjegyzések:

1. Az `aboli` függvény első formális paramétere azért `char*`, mert egy karaktertömböt (stringet) kell átvenni. A második paraméter viszont azért `int*`, mert ennek a paraméternek a segítségével egy értékét akarunk visszaadni a függvényből.

2. A függvény hívás során kihasználtuk azt, hogy egy stringet feldolgozó függvénynek nem csak egy karaktertömb kezdőcímét adhatjuk meg, hanem tetszőleges elemének a címét is. Így intézhettük el, hogy a beolvasott stringnek mindig a még feldolgozatlan részével folytassa a függvény a szám keresését.

3. A második (mutató típusú) formális paraméter helyére aktuális paraméterként a `hossz` változó **címét** írtuk be, így a függvény ezen a címen keresztül a változó **értékét** változtatja meg.

15 Objektumok deklarációja

A deklaráció határozza meg, hogy a C fordító hogyan értelmezze az azonosítókat (azaz milyen objektumok jelölésére használatosak.) A jegyzetben már több helyen volt szó deklarációkról. Ebben a pontban összefoglaljuk a szükséges ismereteket.

15.1 Definició és deklaráció

A definició meghatározza valamely objektum típusát, méretét, és hatására helyfoglalás történik. A definíció egyben deklaráció is.

A deklaráció valamely objektumnak a típusa, mérete (azaz alapvető tulajdonságainak) jelzésére szolgál.

Például:

```
char matrix [10][20]
```
definíció
```
char matrix [][20] deklaráció
```

A teljes programot tekintve minden objektumot **pontosan egyszer** kell definiálni (hacsak nem akarjuk újra definiálni), de lehet, hogy többször kell deklarálni.

15.2 A deklaráció formája

```
[tárolási_osztály] [típusnév] deklarátor_specifikátor
```

A szögletes zárójelek azt jelzik, hogy a `tárolási osztály` és a `típusnév` közül az egyik elmaradhat, ilyenkor a megfelelő alapértelmezés lép életbe.

15.2.1 A típusnév

A deklarációban a típusnév lehet:

- az alaptípusok ismertetésénél felsoroltak valamelyike
- struktúra- és unió definíciók vagy címkék (nevek)
- `typdef`- el definiált típusnevek

15.2.2 A deklarátor specifikátor

A deklarátor specifikátor az alábbiak valamelyike lehet:

Formája	Jelentése
`azonosító`	alaptípus
`azonosító [állandó kifejezés]`	tömb
`azonosító [ ]`	tömb
`azonosító ( )`	függvény
`a fentiek, elôttük *- al`	fenti objektumok mutatói
`(* azonosító ) ()`	függvény-mutató

A fentiek az alábbi korlátozásokkal érvényesek:

1. Tömb csak az alábbiakból képezhető:

   - alaptípusok
   - mutatók
   - struktúrák
   - uniók
   - tömbök

2. Függvényérték nem lehet:

   - tömb
   - unió
   - függvény

de lehet a fentiek bármelyikét megcímző mutató!

Mindezek megértésének megkönnyítésére nézzük az alábbi példákat:

`int t[]`	egészeket tartalmazó tömb
`int *t[]`	egészeket megcímző mutatókat tartalmazó tömb
`int f()`	egészt visszaadó függvény
`int *f()`	egészt megcímzô mutatót visszaadó függvény
`int (*f)()`	egészt visszaadó függvényt megcímző mutató
`int *(f())`	
`int (*f())[]`	olyan tömb, amelyeknek elemei fenti típusú függvény-mutatók

15.2.3 Tárolási osztályok

A definiált objektum érvényességi körét és élettartamát (vagy tárolási módját) határozza meg.

A következő tárolási osztályok léteznek:

`auto`	Lokális változó ("automatikus változó") egy függvényre vagy egy blokkra nézve. Értéke a függvénybe (blokkba) való belépéskor határozatlan, a kilépéskor megszűnik.
`regiszter`	Olyan `auto` változó, amelyet gyakran kívánunk használni. Utasítás a fordítóprogramnak, hogy "könnyen elérhető" módon (például regiszterekben) tárolja az értéküket. Mivel ez a mód gépfüggő lehet, nem alkalmazható rájuk az `&` operátor, hiszen lehet, hogy nincs is valódi értelemben vett címük.
`extern`	Általános érvényű változó, a program különböző részeiben is érvényes.
`static`	Értéke megmarad, nem jön létre és szűnik meg a függvényhívással. (Lehet belső vagy külső.) Érvényességi köre korlátozott.

15.3 Külső és belső azonosítók

Külső definíció az, amely minden függvényen kívül helyezkedik el. A függvénydefiníció mindig külső.

A C program külső definíciók sorozata. A külső definícióban az `extern` és `static` tárolási osztály használható. Az alapértelmezés az `extern` tárolási osztály.

Az egyes függvényekben `extern` deklarációval jelezni lehet a külső változókat. Ez azonban csak akkor kötelező, ha a deklaráció egy forrásszövegben megelőzi a definíciót, vagy a definíció más forrásállományban van, mint a függvény.

A gyakorlottabb programozók által használt konvenció: valamennyi külső definíciót helyezzük el a forrás-file elején, és ne használjunk `extern` deklarációt a függvényeken belül.

Belső definíció az, amely függvényeken vagy blokkon belül helyezkedik el. Tárolási osztálya lehet `auto` (feltételezett), `register` vagy `static`.

15.4 Az érvényességi tartomány szabályai

Egy C program több forrás-file-ból állhat, és könyvtárakban előre lefordított rutinokra is hivatkozhat. Ezért szükséges tisztázni az azonosítók érvényességi tartományát.

Kétféle érvényességi tartományról beszélhetünk:

- lexikális érvényességi tartomány
- a külső változók érvényességi tartománya

15.4.1 Lexikális érvényességi tartomány

A programnak az a része, amelyben "definiálatlan azonosító" hibajelzés nélkül használhatjuk az azonosítót. Részletesebben:

1. *Külső definíciók*: a definíciótól az őket tartalmazó forrásállomány végéig.

2. Formális paraméter: az a függvény, amelynek fejében szerepel.

3. Belső definíció: az a függvény vagy blokk, amelyben szerepel.

4. Címke: az a függvény, amelyben előfordul. (Nincs "külső" címke!)

Megjegyzés

Ha egy azonosítót egy blokk vagy függvény fejében explicit módon deklarálunk, akkor annak végéig az adott azonosító összes blokkon kívüli deklarációja felfüggesztődik (*újradefiniálás*).

15.4.2 A külső azonosítók érvényességi tartománya

A program egészére nézve tisztázza, hogy ugyanarra az azonosítóra vonatkozó hivatkozás ugyanazt az objektumot jelenti-e.

Egy `extern`-ként deklarált változónak a teljes programot alkotó forrásállományok és könyvtárak valamelyikében pontosan egyszer definiáltnak kell lennie. Minden rá hivatkozó függvényt tartalmazó állományban (esetleg magában a függvényben - bár ez nem szokásos) szerepelnie kell az azonosító deklarációjának. Így ezzel az azonosítóval minden függvény ugyanarra az objektumra hivatkozik. A deklarációknak ezért kompatibiliseknek kell lenniük. (Típus és méret szempontjából.)

A legfelső szinten `static`-ként deklarált azonosító csak az adott állományra érvényes, a többi állományban szereplő függvények nem ismerik. Függvény is deklarálható `static`-ként.

15.5 Implicit deklarációk

Ebben a pontban ismertetjük azokat az alapértelmezéseket, amelyek a deklarációkkal kapcsolatosak.

A tárolási osztály alapértelmezése:

- külső definíciókban: `extern`
- függvényen belül: `auto`

Típus alapértelmezése: `int`.

Nem hiányozhat egyszerre a tárolási osztály és a típus.

Kifejezésekben azt a nem deklarált azonosítót, amelyet "(" követ, a fordító `int`- et visszaadó fügvénynek értelmezi. (Ezért ilyeneknek a deklarációja elhagyható - bár a helyes programozási stílus *megköveteli* a függvények deklarálását, azaz a prototípusok alkalmazását.)

15.6 Az inicializálás

A deklarációban (bizonyos korlátozásokkal) az objektumoknak kezdőérték adható. Inicializálás nélkül:

- a külső és a statikus változók értéke garantáltan 0
- az automatikus és regisztervváltozók értéke határozatlan.

Egyszerű változó inicializálása:

```
tár_o típus azonosító = kif
```

Összetett objektum inicializálása:

```
deklaráció = { kif, kif,....,}
```

A kifejezések az elemek sorrendjében adódnak át. Ha számuk kevesebb, mint amit a deklaráció meghatároz a maradék elemek 0-val töltődnek fel. Ha több, hibajelzést kapunk.

Tömbök esetén a deklarációból az első index felső határa elmaradhat, ezt ekkor a kifejezések száma határozza meg.

15.6.1 Külső és statikus változók inicializálása

Az inicializálás egyszer, fordítási időben, a helyfoglalással együtt történik. A kifejezés lehet

- állandó kifejezés
- mint az állandó kifejezés, de szerepelhet operandusként már deklarált változó címe (+ eltolás) is.

15.6.2 Automatikus és regiszter változók inicializálása

Minden alkalommal végrehajtódik, amikor a vezérlés belép a blokkba. A kifejezés tetszőleges, korábban definiált értéket tartalmazhat. (Változókat, függvényhívásokat stb.) Valójában rövidített formában írt értékadás.

Példák:

```
int t[] = {0,1,2,13,26,45}
int m[5][3] = {1,3,5,2,4,6,7,9,11,8,10,12,13,15,17}
```

15.6.3 Karaktertömb inicializálása

Karaktertömb inicializálására használható karakterlánc is, ezzel egyszerűsíthető a kezdőértékadás.

Példa:

```
char s[] = "szoveg";
```

ami egyenértékű a "szabályos"

```
char s[] = {'s','z','o','v','e','g','\0'};
```

formával.

16. Fomattált beolvasás

A standard inputról formattált beolvasást végez a `scanf` függvény. Formája:

```
scanf("Konverziós karakterek", p1, p2, ...)
```

A függvény a konverziós karakterek által megadott módon adatokat olvas be, majd azok konvertálással kapott értékét sorban a paraméterekhez rendeli. A `p1, p2, ...` paramétereknek mutatóknak kell lenniük. A pontos működése meglehetősen bonyolult, itt csak egy egyszerűsített leírását adjuk meg.

A függvény adatok és üreshely karakterek (helyköz, tabulátor, új sor, kocsi vissza, soremelés, függőleges tabulátor és lapemelés) váltakozásaként tekinti a bemenetet. Az üreshely-karaktereket átlépi, a köztük levő karaktereket pedig konvertálja és az így kapott értéket az aktuális paraméter által megcímzett változóban tárolja.

A konverziós karakterek az alábbiak lehetnek (a felsorolás nem teljes!):

Konverzió	Az argumentum típusa	A beolvasott adat
`%d`	`int*`	Egész szám
`%f`	`float*`	Valós szám
`%lf`	`double*`	Valós szám
`%c`	`char*`	Egy karakter
`%s`	`char*`	Karaktersorozat (a záró 0-át elhelyezi)

Ügyeljünk a használat során arra, hogy a konverziós karakterek között más karakterek ne legyenek (még helyközök sem), mert azoknak is van jelentésük, amelyet itt most nem részletezünk.

16.1. Példaprogam: egy valós tömb elemeinek beolvasása és rendezése

A feladat egy valós tömb elemszámának és elemeinek beolvasása és a tömb rendezése a kiválasztásos és a buborék rendezés segítségével.

```
/*  CPELDA7.C  */
/* Készítette: Ficsor Lajos */

/* A program beolvas egy double tömböt és rendezi kétféle
   módszerrel */
#include <stdio.h>

#define MAXELEM 100

void rendez_cseres(double* tomb, int elemszam);
void rendez_buborek(double* tomb, int ele4mszam);

void main(void)
{
int i,n;
double a[MAXELEM];

/* Elemszam beolvasasa */
scanf ("%d",&n);

/* Tombelemek beolvasasa */
for (i=0; i<n; i++) scanf("%lf",a+i);

rendez_cseres (a,n);
for (i=0; i<n; i++) printf("%lf ",a[i]);
printf("\n");

rendez_buborek (a,n);
for (i=0; i<n; i++) printf("%lf ",a[i]);
printf("\n");

}
```

```
    void rendez_kiv(double* tomb, int elemszam)
    /* A függvény növekvő sorrendbe rendezi a tömb tömböt */
    {
    int i,j,minindex;
    double min, seged;

    for (i=0; i<elemszam-1; i++)
      {
      /* legkisebb elem keresese az aktualis elemtol a tomb
         vegeig */
      min = tomb[i];
      minindex = i;
      for (j=i+1; j<elemszam; j++)
        {
        if (tomb[j] < min)
          {
          min = tomb[j];
          minindex = j;
          }
        }
      /* Ha nem az aktualis elem a legkisebb, csere */
      if (minindex != i)
        {
        seged = tomb[i];
        tomb[i] = min;
        tomb[minindex] = seged;
        }
      }

    }

    void rendez_buborek(double* tomb, int elemszam)
    /* A függvény növekvő sorrendbe rendezi a tömb tömböt */
    {
    int i,csere;
    double seged;

    csere =1;
    while (csere)    /* Mindaddig, amíg csere szükséges volt */
      {
      csere = 0;
      for (i=0; i<elemszam-1; i++)
        if (tomb[i] > tomb[i+1])    /* Ha a szomszédos elemek */
          {                         /* sorrendje nem jó       */
          seged = tomb[i];          /* Csere */
          tomb[i] = tomb[i+1];
          tomb[i+1] = seged;
          csere++;
          }
      }

}
```

16.2. Példaprogram: új elem beszúrása rendezett tömbbe

A feladat egy program írása, amely a standard inputról pozitív egész számokat olvas, és ezeket elhelyezi nagyság szerint növekvő sorrendben egy tömb egymás után következő elemeibe. A beolvasás végét az első nem pozitív szám jelzi.

```
/*  CPELDA10.C  */
/* Készítette: Ficsor Lajos */

/* A program a standard inputról pozitív egész értékeket
   olvas be, és egy tömbben nagyság szerint növekvő
   sorrendben helyezi el. A beolvasás végét az első nem
   pozitív szám jelzi.
*/

#include <stdio.h>

#define MAX 100
#define JOKICSISZAM 0

void main(void)
{
int tomb[MAX+1]; /* A tomb, amely a 1. indexu elemetol */
                 /*kezdve tartalmazza a beolvasott szamokat */
int elemek;      /* A tombben elhelyezett elemek szama */
int szam,i;

elemek = 0;
/* A tomb 0. elemebe elhelyezunk egy olyan kis szamot,
   amely az adatok kozott nem fordulhat elo */
tomb[0] = JOKICSISZAM;

scanf("%d",&szam);
while (szam > 0)
  {
  /* A tomb vegetol kezdve minden elemet eggyel hatrebb
     (nagyobb indexu helyre) helyezunk, amig a beszurando
     szamnal kisebbet nem talalunk. Ekkor az uj elemet
     ez utan az elem utan tesszuk be a tombbe.
     Az ures tombre is jol mukodik.
  */
  for (i=elemek; i>0 && szam<tomb[i]; i--)
    {
    tomb[i+1] = tomb[i];
    }
  tomb[++i] = szam;
  elemek++;
  printf ("A tomb elemszama: %d\n",elemek);
  for (i=0; i<=elemek; i++) printf (" %d",tomb[i]);
  printf("\n");
```

```
  /* Kovetkezo szam beolvasasa */

  scanf("%d",&szam);
  } /* while szam > 0 */
printf ("\nA vegleges tomb:\n");
for (i=1; i<=elemek; i++) printf (" %d",tomb[i]);
}
```

17. Több forrásfile-ból álló programok

Egy C program a gyakorlatban mindig több forrásfile-ból áll, mert a túlságosan hosszú szövegek nehezen kezelhetők. Ezzel lehetővé válik az is, hogy egy forrásfile-ban csak a logikailag összetartozó függvények definíciói legyenek.

Ha azonban egy függvény definíciója és hívása nem azonos forrásfile-ban van, a hívást tartalmazó file-ban azt deklarálni kell. A szükséges deklarációk nyilvántartása nem egyszerű feladat, különösen ha figyelembe vesszük, hogy egy változtatást minden helyen át kell vezetni.

Ugyanez a probléma akkor, ha könyvtárban tárolt (például szabványos) függvényeket hívunk. Ezeknek a deklarációját is meg kell adnunk a hívást tartalmazó forrásban.

A probléma megoldására szolgál az
```
#include <filenév>
```
vagy
```
#include "filenév"
```
direktíva. Jelentése: a `filenév` által meghatározott file tartalma a direktíva helyén bemásolódik a forrásfile-ba. Az első esetben a bemásolandó file-t a fejlesztő környezetben beállított szabványos helyen, a második forma esetén az aktuális katalógusban keresi a rendszer.

Ezt a direktívát használhatjuk fel a deklarációk egyszerűsítésére. Az elterjedt programozói gyakorlat szerint minden forrásfile (szokásos kiterjesztése: `.C`) csak a függvények definícióit tartalmazza, a deklarációikat egy ugyanolyan nevű. de `.H` kiterjesztésű file-ba (header file) gyűjtjük össze. A definíciókat tartalmazó forrásfile-ba ez a header file egy `#include` direktíva segítségével kerül be, és ugyanezt az eljárást használjuk a hívásokat tartalmazó forrásfile-ok esetén is.

Az eddigi mintapéldák elején található
```
#include <stdio.h>
```
sor magyarázata az. hogy a szabványos input-output függvények deklarációit a (szabványos) `stdio.h` header file tartalmazza. A függvények leírását tartalmazó dokumentáció minden függvényre megadja, hogy milyen nevű header file tartalmazza az adott függvény deklarációját.

A header file-ok a függvény-deklarációkon kívül egyéb deklarációkat, szimbolikus konstansokat stb. is tartalmazhatnak. Például az eddigi példákban is már használt `EOF` az `stdio.h` file-ban deklarált szimbolikus konstans.

17.1. Példaprogram: egy szöveg sorainak, szavainak és karaktereinek száma

Írjunk programot, amely meghatározza a standard inputról beolvasott file sorainak, szavainak és karaktereinek számát. Szó: nem helyközzel kezdődő, helyközzel vagy sor végével végződő karaktersorozat. A file-t soronként olvassuk be, a már előzőleg megírt `getstr` függvény segítségével.

Bár a feladat mérete ezt most nem indokolja, gyakorlásképpen a főprogramot és a függvényeket (a már ismert `getstr` és a most megírandó `egyszo` függvényt) tegyük külön forrásfile-okba. Legyen tehát az `STR.C` file-ban a két függvény definíciója, deklarációik pedig az `STR.H` file-ban. A főprogram forrásfile-ja legyen a `SZAVAK.C` .

```
/* CPELDA8 */
/* Készítette: Ficsor Lajos */
```

STR.H

```
 int getstr (char* s);
 void strmasol (char* cel, char* forras);
```

STR.C

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

int getstr (char* s)
/*****************************************************
  A fuggveny egy sort olvas be a standard inputrol,
  es az s tombbe helyezi el, string-kent.
  Visszateresi erteke a beolvasott karakterek szama,
  vagy EOF
*****************************************************/
{
int c;
int i;

i=0;
while ( (c=getchar()) != '\n' && c !=EOF)
  {
  s[i++] = c;
  }
s[i] = '\0';
return c==EOF ? c : i;
}


int egyszo(char*s, char* szo, int* szohossz)
/*****************************************************
   A fuggveny beolvas az "s" stringbol egy szot, es
   elteszi a "szo" stringbe, a szo hosszat pedig a
   "szohossz" parameterbe.
   Szo definicioja: nem helykozzel kezdodo, helykozel
   vagy string vegevel hatarolt karaktersorozat. A string
```

```
   elejetol kezdi a keresest.
   Fuggvenyertek: az "s" stringbol feldolgozott karakterek
   szama.
*********************************************************/
{
int i;

i=0;
*szohossz = 0;
while ( s[i] == ' ') i++;  /* Szokozok atlepese, a szo */
                           /* elejenek keresere        */
while ( s[i] != ' ' && s[i] != '\0') /* Szo belsejeben */
   { szo[(*szohossz)++] = s[i++]; }  /* vagyunk */
szo[*szohossz] = '\0';          /* Zaro 0 elhelyezese */
return i;
}
```

/* SZAVAK.C */

```
/* Keszitette: Ficsor Lajos */

/*********************************************************
   A program a standard inputrol beolvasott szovegfile
   sorainak, szavainak es karaktereinek szamat hatarozza
   meg.
   Ismert hiba:
   Ha az utolso sor nem sorveggel, hanem EOF-el er veget,
   figyelmen kivul marad!
*********************************************************/

#include <stdio.h>
#include "str.h"

#define MAXSORHOSSZ 200
#define MAXSZOHOSSZ 200

void main(void)
{
char sor[MAXSORHOSSZ+1];
char szo[MAXSZOHOSSZ+1];
int sorszam, szoszam, karakterszam;
int sorhossz, szohossz;
int kezdet;
int i;

sorszam=0;
karakterszam=0;
szoszam=0;
/* Soronkenti beolvasas a file vegeig */
while ( (sorhossz=getstrl(sor)) != EOF)
  {
  sorszam++;
  karakterszam += sorhossz;
  /* A sor szetszedese szavakka */
```

```
    kezdet = 0;
    while (kezdet < sorhossz)
      {
      i = egyszo(sor+kezdet, szo, &szohossz);
      kezdet += i;
      if (szohossz) szoszam++;  /* Ha valodi szo */
      }
    }

  printf ("Sorok szama:            %d\n",sorszam);
  printf ("Szavak szama:           %d\n",szoszam);
  printf ("Karakterek szama szama: %d\n",karakterszam);

  }
```

18. A struktúra és az unió

18.1 A struktúra deklarációja

A struktúra olyan összetett típus, amelynek elemei különböző típusúak lehetnek. (A tömb azonos típusú elemek összefogására alkalmas!)

Deklarációja az alábbi formák valamelyikével lehetséges:

```
struct {elemek deklarációja} változólista
```

Deklarálja, hogy a `változólista` elemei adott szerkezetű struktúrák.

```
struct név {elemek deklarációja}
```

Létrehoz egy `név` nevű típust, amely a megadott szerkezetű struktúra. Ezután a `struct név` típusnévként használható deklarációkban. Ez a forma az ajánlott.

Az elemek deklarációja pontosvesszővel elválasztott deklarációlista.

Példák:

```
stuct {
  int év;
  charhonap[12];
  int nap;
  } szulinap, ma, tegnap;
```

vagy

```
struct datum {
  int év;
  char honap[12];
  int nap;
```

```
 };
struct datum szulinap, ma, tegnap,*pd;
```

Struktúra tagja lehet már definiált struktúra (vagy annak mutatója) is, de nem lehet tagja önmaga. Lehet viszont tagja önmaga mutatója (önhivatkozó struktúra).

18.2 Hivatkozás a struktúra elemeire

A struktúrákkal az alábbi műveleteket végezhetjük:

- az `&` operátorral képezhetjük a címét
- hivatkozhatunk valamelyik elemére a "." (pont) és a "`->`" operátorral
- megengedett két azonos típusú struktúra közötti értékadás
- struktúra lehet függvény paramétere vagy visszatérési értéke

A struktúra elemére (tagjára vagy mezőjére) hivatkozhatunk a

```
struktúra_azonosító.tagnév
```

konstrukcióval. Például:

```
szulinap.ev = 1951;
```

A hivatkozás történhet struktúra-mutató segítségével is. Mivel a C nyelvben ez a forma gyakori, erre külön operátor is van. A hivatkozás formája ebben az esetben:

```
(*pd).név
```

vagy

```
pd -> nev
```

A fenti két forma egyenértékű.

18.3 Struktúra - tömbök

A struktúra-típus lehet tömb alaptípusa. Például:

```
struct datum napok[20]
```

18.4. Unió

Az unió olyan változó, amely (különböző időpontban) különféle típusú és méretű objektumokat tartalmazhat, ezáltal ugyanazon tárterületet különféleképpen használhatunk.

Az unió deklarációja formálisan ugyanolyan, mint a struktúráé, de a `struct` alapszó helyett a `union` alapszót kell használni, azonban míg a struktúrában az elemek felsorolása az adott elemek

sorozatát jelenti, az unionban a felsorolás "az alábbiak egyike" értelmű. A programozó felelősége, hogy következetesen használja ezt a konstrukciót.

Az alábbi kis példaprogram segítségével illusztráljuk a unió és a struktúra használatát. A `datum` struktúra első eleme egy egész változó, amellyel jelezni lehet, hogy a második eleme, amely egy unió, a lehetséges értékek közül éppen milyen típusút használ.

```
#include <stdio>

main() {

struct angol_datum {
  int day;
  int month;
  int year;};

struct usa_datum {
  int mm;
  int dd;
  int yy; };

struct magyar_datum {
  int ev;
  char honap[12];
  int nap; };

struct datum {
  int tipuskod;
  union {
    struct angd d1;
    struct usad d2;
    struct magyd d3;}dat;
};

struct datum a,b,c;

a.tipuskod = 1;
a.dat.d1.day = 10;
a.dat.d1.month = 9;
a.dat.d1.year = 1951;
printf ("Angol datum: %d %d %d \n",
        a.dat.d1.day,a.dat.d1.month,a.dat.d1.year);

a.tipuskod = 2;
a.dat.d2.dd = 10;
a.dat.d2.mm = 9;
a.dat.d2.yy = 1951;
printf ("USA datum: %d %d %d \n",
        a.dat.d2.mm,a.dat.d2.dd,a.dat.d2.yy);

a.tipuskod = 3;
a.dat.d3.nap = 10;
a.dat.d3.honap = "szeptember";
```

```
a.dat.d3.ev = 1951;
printf ("Magyar datum: %d %s %d \n",
         a.dat.d3.ev,a.dat.d3.honap,a.dat.d3.nap);

}
```

18.5. Példaprogram: szó-statisztika (1. változat)

Ebben az alpontban a cél egy összetettebb példa megoldása: határozzuk meg az input szövegfile szavainak gyakoriságát!

Első lépés: a szavak elhatárolása. Ehhez a CPELDA8 program *egyszo* függvényének továbbfejlesztése szükséges, a szó-határoló karakterek pontos meghatározásával. A továbbfejlesztett változat a NYTSZO.C file-ban található.

Második lépés: a szavak tárolási módszerének meghatározása. Az első ötlet lehet két, párhuzamosan kezelendő tömb:

```
char szavak[MAXSZOHOSSZ+1][MAXDB]
int db[MAXDB]
```

Ez nem túl jó megoldás, mert logikailag összetartozó adatok különböző adatszerkezetekben találhatók.

Jobb megoldás: struktúratömb használata. A szavak tárolására szolgáló adatszerkezet így:

```
struct szodb {
   char szo[MAXSZOHOSSZ+1];
   int db;
   };

struct szodb szavak[MAXDB];
```

Ennek az adatszerkezetnek is van hátránya:

- fölösleges helyfoglalás, mert a maximális szóérethez kell lefoglalni a helyet
- fölösleges helyfoglalás, mert a maximális szószámhoz kell lefoglalni a helyet
- mivel keresésre lesz szükség, a szavakat rendezetten kell tartani, ami sok adatmozgatást igényel

A későbbiekben a fenti adatszerkezetet úgy fogjuk módosítani, hogy a fenti hátrányok csökkenjenek, most azonban fogadjuk el ezt a megoldást.

Az algoritmus az alábbi lehet:

1. Olvassuk végig soronként a file-t, amíg a file végét el nem érjük
    2. Minden sort vágjunk szét szavakra
        3. Minden szót keressünk meg az eddigi szavakat tároló táblázatban (a SZAVAK tömbben)
           Ha megtaláltuk, a gyakoriságot növeljük eggyel
           Ha nem találtuk meg, szúrjuk be a tömbbe, 1 gyakorisággal.

Az 1. és 2. pontban leírt műveletek a CPELDA8 programban kialakítottak szerint megoldhatók, tehát ennek a programnak a módosításával lehet a legkönnyebben a feladatot megoldani.

A 3. pont keresési műveletét a CPELDA10 programban implementált algoritmus finomításával implementálhatjuk. Az előfordulás vagy beszúrás helyének megkeresését különválasztva erre egy *keres* nevű függvényt írunk, a beszúráshoz tartozó adatmozgatást a főprogramra bízzuk.

Mivel a *keres* nevű függvény és a főprogram egyaránt használja a *szodb* struktúrát, ennek deklarációját **külső** deklarációként oldjuk meg, és a konstans definíciókkal együtt a *SZODEF1.H* header file-ba telepítjük.

Ezek alapján a program az alábbi modulokat tartalmazza:

- `SZODEF1.C` amely hivatkozik a `SZODEF1.H`, az `STR.H`, az `NYTSZO.H` és a `SZOFGV1.H` header file-okra
- `NYTSZO.C`
- `SZOFGV1.C`, amely hivatkozik a `SZODEF1.H` header file-ra
- `STR.C`, amely korábban lett kifejlesztve, és a soronkénti beolvasáshoz szükséges `getstrl` függvény miatt szükséges.

```
/*  NYTSZO.C  */
/* Keszitette: Ficsor Lajos */

#include "nytszo.h"

int nytszo(char*s, char* szo, int* szohossz)
/*******************************************************
   A fuggveny beolvas az "s" stringbol egy szot, es
   elteszi a "szo" stringbe, a szo hosszat pedig a
   "szohossz" parameterbe.
   Szo definicioja: nem hatarolo karakterrel kezdodo, hatarolo
   karakterrel vagy string vegevel vegzodo karaktersorozat.
   A string elejetol kezdi a keresest.
   Hatarolo karakterek: amit a "hatarolo" fgv. annak jelez.
   Fuggvenyertek: az "s" stringbol feldolgozott karaktere
   szama.

   Hivott fuggvenyek: hatarolo
*******************************************************/
{
int i;

i=0;
*szohossz = 0;
while ( hatarolo(s[i])) i++; /* Szo elejenek keresese */
while ( !hatarolo(s[i])  && s[i] != '\0')
   { szo[(*szohossz)++] = s[i++]; }
szo[*szohossz] = '\0';
return i;
}
```

```
int hatarolo(char c)
/*
{
int hatar;

hatar = 0;
if (c==' ' || c=='\t' || c=='.' || c==','
    || c== '(' || c==')' || c=='!' || c =='?') hatar = 1;
return hatar;
}

Persze ez irhato sokkal rovidebben:   */

{
return c==' ' || c=='\t' || c=='.' || c==',' || c==':'
       || c==';' || c== '(' || c==')' || c=='!' || c =='?'
       || c=='"';
}
```

/* SZODEF1.H */

```
/* Definiciok a szo-statisztikahoz.
   1. valtozat
*/
#define MAXSORHOSSZ 300
#define MAXSZOHOSSZ 50
#define MAXDB 800

struct szodb {
  char szo[MAXSZOHOSSZ+1];
  int db;
  };
```

/* SZOFGV1.C */

```
/* Keszitette: Ficsor Lajos */

/* A szo statisztika keszitesehez szukseges fuggvenyek */

#include <string.h>
#include "szodef1.h"
#include "szofgv1.h"

int keres (struct szodb tabla[],   /* A szavak tablazata */
     int tablahossz,         /* A tabla elemeinak szama */
     char* szo,              /* A keresendo szo */
     int* bennevan)          /* =1, ha a szot megtalaltuk,
                                =0 ha nem                */
/* A fgv. a "szavak" tombben keresi a "szo" szot.
   Visszateresi ertek:
```

```
  a keresett szo sorszama (vagy helye) a tombben
*/
{
int i,j;
int hely = 0;
int talalt = 0;

*bennevan = 0;

for (i=0; !talalt && i<=tablahossz; i++)
  {
  j = strcmp(szo, tabla[i].szo);
  if (j<=0)
    {
    hely = i;
    talalt = 1;
    if ( j == 0) *bennevan = 1;
    }
  }
return hely;
}
```

/* SZOSTAT1.C */

```
/* Szostatisztika program
   1. valtozat */

/* Keszitette: Ficsor Lajos */

#include <stdio.h>
#include <string.h>
#include "szodef1.h"
#include "str.h"
#include "nytszo.h"
#include "szofgv1.h"

void main(void)
{
char sor[MAXSORHOSSZ+1];
char szo[MAXSZOHOSSZ+1];
struct szodb szavak[MAXDB+1];
char* soreleje;
int sorhossz, szohossz;
int kezdet;
int hely;
int tablahossz;
int bennevan;

int i;

/* Szo-tabla elokeszitese */
szavak[0].szo[0] = 'z'+1;
szavak[0].szo[1] = '\0';
```

```
tablahossz = 0;

/* Soronkenti beolvasas a file vegeig */
while ( (sorhossz=getstrl(sor, MAXSORHOSSZ)) != EOF)
  {
  /* A sor szetszedese szavakka */
  kezdet = 0;
    while (kezdet < sorhossz)
    {
    i = nytszo(sor+kezdet, szo, &szohossz);
    kezdet += i;
    if (szohossz)
      {
      /* Valodi szot talalt */
      printf ("\n%s",szo);
      /* Kereses a tablaban */
      hely = keres (szavak, tablahossz, szo, &bennevan);
      if (bennevan)
        {
        /* Megtalalta */

        szavak[hely].db++;
        }
      else
        {
        /* Nem talalta meg. Betesszuk a tablaba */
        for (i=tablahossz; i>=hely; i--)
          {
          szavak[i+1] = szavak[i];
          }
        strcpy(szavak[hely].szo,szo);
        szavak[hely].db = 1;
        tablahossz++;
        }  /* If bennevan */
      printf ("\n%d %d %s\n",tablahossz, hely,
              szavak[hely].szo);
      for (i=0; i<tablahossz; i++)  /* Csak ellenorzo kiiras */
        {
        printf ("\n%50s %5d\n", szavak[i].szo, szavak[i].db);
        }

      } /* If szohossz */
    } /* while kezdet < sorhossz */
  } /* while ...!-EOF */
/* Statisztika kiirasa */
for (i=0; i<tablahossz; i++)
  {
  printf ("\n%50s %5d", szavak[i].szo, szavak[i].db);
  }
printf("\n");
}
```

19. Dinamikus memória kezelés

Ha egy változót vagy tömböt definiálunk, annak hely foglalódik a memóriában, ami az adott változó vagy tömb teljes élettartama alatt rendelkezésre áll, akár kihasználjuk, akár nem. A tömbök esetén a méretet konstans kifejezésként kell megadnunk, tehát az nem függhet a program közben meghatározott értékektől.

Mindkét problémát megoldja a dinamikus memóriafoglalás lehetősége. Segítségével csak akkor kell lefoglalni az adatok tárolásához a helyet, amikor azt használni akarjuk, és csak annyit, amennyi szükséges. Ha az adatokra már nincs szükség, a lefoglalt memória felszabadítható és más célra újra lefoglalható.

A lefoglalt memória a kezdőcímét tartalmazó pointer segítségével kezelhető.

19.1 Memóriakezelő függvények

Használatukhoz szükséges az `stdlib.h` header file.

```
void* malloc(size_t meret)
```

Lefoglal `meret` byte nagyságú memóriaterületet. Visszatérési értéke a lefoglalt terület kezdőcíme, vagy `NULL`, ha a helyfoglalás sikertelen volt.
A visszaadott pointer `void*` típusú, ha tehát a memóriaterületet tömb-szerűen akarjuk használni (mint általában szokás), azt a tárolni kívánt adatok típusára kell konvertálni.
A `size_t` előre definiált típus (olyan egész, amelyben biztosan elfér a lefoglalandó terület mérete), `NULL` pedig a 0 mutató számára definiált szimbolikus konstans.

```
void free(void* blokk)
```

Felszabadítja a `blokk` kezdőcímű memóriaterületet. A paraméterének előzőleg `malloc`, `calloc` vagy a `realloc` függvény segítségével kellett értéket kapnia.

```
void* calloc (size_t db, size_t meret)
```

Lefoglal `db*meret` byte nagyságú memóriaterületet. Visszatérési értéke mint az `malloc` függvénynél.

```
void* realloc(void* blokk, size_t meret)
```

A `blokk` egy már korábban lefoglalt terület kezdőcíme. Ennek a területnek a méretét `meret` nagyságúra állítja. Ha a méret csökken, adatok vesznek el. Ha nő, a terület kezdőcíme (ami a visszatérési érték) megváltozhat. Ekkor a korábbi adatok az új helyre másolódnak.

19.2. Példaprogram: szó-statisztika (2. változat)

A program első változatában alkalmazott adatszerkezet minden szó számára valamilyen maximális szóhossznak megfelelő helyet foglalt le. Ez fölösleges memóriapocséklás, ráadásul a program emiatt a `MAXSZOHOSSZ`-nál hosszabb szavakat nem tudja kezelni.

A feladat most a program módosítása úgy, hogy a helyfoglalása kedvezőbb legyen. Ehhez a szavak adatait tartalmazó struktúrát átalakítjuk úgy, hogy ne a szót, hanem annak pointerét tartalmazza. Minden szó számára dinamikusan foglalunk helyet, annyit, amennyi szükséges.

Az új struktúra:

```
struct szodb {
   char* szo;
   int db;
}
```

Az átalakítás miatt meglepően kevés helyen kell módosítani a programot! (Ez is mutatja a pointer hasznát a C nyelvben.)

Továbbra is fölösleges helyfoglalást okoz az, hogy a struktúra-tömböt fix mérettel kell deklarálni. Ezt csak úgy kerülhetjük el, hogy más adatszerkezetet használunk. A feladat legelegánsabb megoldása a *bináris fa* adatszerkezet segítségével készíthető el. Ez egyben a beszúráshoz szükséges keresésre is jó megoldást ad.

```
/* SZODEF2.H */
/* Definiciok a szo-statisztikahoz.
   2. valtozat
*/
#define MAXSORHOSSZ 300
#define MAXSZOHOSSZ 50
#define MAXDB 100

struct szodb {
  char* szo;
  int db;
  };
```

```
/* SZOSTAT2.C */

/* Szostatisztika program
   2. valtozat */

/* Keszitette: Ficsor Lajos */

#include <stdio.h>
#include <string.h>
#include <alloc.h>
#include "str.h"
#include "nytszo.h"
#include "szodef2.h"
#include "szofgv2.h"

Mindössze két helyen van változás az 1. változathoz képest:


/* Szo-tabla elokeszitese */
```

```
/* Itt a különbseg!*/
szavak[0].szo = (char*) malloc(MAXSZOHOSSZ+1);
/* Innentől már változatlan /*
szavak[0].szo[0] = 'z'+1);
szavak[0].szo[1] = '\0';
tablahossz = 0;

.
.
.

        {
        /* Nem talalta meg. Betesszuk a tablaba */
        for (i=tablahossz; i>=hely; i--)
          {
          szavak[i+1] = szavak[i];
          }
        /*Masik kulonbseg!! */
        szavak[hely].szo = (char*) malloc(strlen(szo)+1);
        /* Innen újra változatlan! */
        strcpy(szavak[hely].szo,szo);
        szavak[hely].db = 1;
        tablahossz++;
        }  /* If bennevan */



/* SZOFGV2.C */
/* 2. valtozat */

/* Keszitette: Ficsor Lajos */

/* A szo statisztika keszitesehez szukseges fuggvenyek
   Csak a struktúra definícióját tartalmazó szodef2.h
   miatt változik, maga a függvény azonos!
*/

#include <string.h>
#include "szodef2.h"
#include "szofgv2.h"
```

A teljes változat a lemezmellékleten megtalálható.

20. Parancssor-argumentumok

Ha az általunk megírt programot parancssorból indítjuk el, van lehetőségünk arra, hogy a parancssorban adatokat adjunk meg számára.

A *main* függvény prototípusa az eddigi példákban

```
void main(void)
```

volt. Teljes alakja azonban

```
int main(int argc, char* argv[], char* env[])
```

A `main` függvény visszatérési értékét a programot aktivizáló program (általában az operációs rendszer) kapja meg. A 0 visszaadott érték megállapodás szerint a sikeres végrehajtást jelenti. Az ettől eltérő értékkel a program befejeződésének okára utalhatunk. A visszatérési értéket a `main` függvényben levő `return` utasítás, vagy a bárhol meghívható `exit` függvény állíthatja be.

Az `exit` függvény használatához az `stdlib.h` vagy `process.h` fejlécfile szükséges. Prototípusa:

```
void exit(int status)
```

A `main` függvény első két paramétere a program aktivizálásához használt parancssorban található paramétereket adja vissza, a harmadik (ami el is maradhat) pedig az operációs rendszer környezeti változóit tartalmazza, az alábbiak szerint:

`argc` a paraméterek száma + 1

`argv[0]` a parancssorban a prompt után gépelt első stringre (a program neve, esetleg elérési úttal) mutató pointer

`argv[1]` az első paraméterre mutató pointer

.

.

.

`argv[argc-1]` az utolsó paraméterre mutató pointer

`argv[argc] = NULL`

Hasonlóan az `env[0], env[1], ... , env[n-1]` a program hívásakor érvényes környezeti változókat tartalmazza, (ahol n a környezeti változók száma), `env[n]=NULL`.

A parancssor paramétereit helyköz választja el. Ha egy paraméterben helyköz lenne, azt idézőjelek közé kell írni. (A program az idézőjelet nem kapja meg!)

21. Szabványos függvények

A C nyelv használatát számos szabványos függvény segíti. Ezek ismertetésére itt természetesen nincs hely, ezért csak felsoroljuk a legfontosabb témacsoportokat, és azokat a header file-okat, amelyet a szükséges deklarációkat és egyéb elemeket tartalmazzák.

Nem teljes körűen ismertetjük a string kezeléshez és a file kezeléshez szükséges függvényeket.

21.1. Szabványos header file-ok

Header file neve	Funkció
`stdio.h`	Szabványos adatbevitel és adatkivitel
`ctype.h`	Karakter-vizsgálatok
`string.h`	String kezelő függvények
`math.h`	Matematikai függvények
`stdlib.h`	Kiegészítő rendszerfüggvények
`assert.h`	Programdiagnosztika
`setjmp.h`	Nem lokális vezérlésátadások
`time.h`	Dátum és idő kezelése
`signal.h`	Jelzések (UNIX signal-ok) kezelése
`limits.h`	A gépi megvalósításban definiált határértékek
`float.h`	
`stdarg.h`	Változó hosszúságú argumentumlisták kezelése

Természetesen minden gépi megvalósítás további, rendszer-specifikus függvényeket is tartalmaz. (Ilyen például DOS alatt a `conio.h` a direkt képernyőkezelés függvényeivel.)

21.2. String kezelő függvények

Itt ismertetjük a legfontosabb karakterkezelő függvényeket. A felsorolás nem teljes!

21.2.1. Karakterátalakító függvények.

Szükséges a `ctype.h` header file.

```
int tolower(int ch)
```

A paraméterét kisbetűsre konvertálja, ha az az A - Z tartományba esik, változatlanul hagyja egyébként.

```
int toupper(int ch)
```

A paraméterét nagybetűsre konvertálja, ha az az a - z tartományba esik, változatlanul hagyja egyébként.

21.2.2. További string-kezelő függvények:

Szükséges a `string.h` header file.

```
int strcmp(char* s1, char*s2)
```

Összehasonlítja az paramétereit, és a visszatérési értéke

negatív, ha s1 < s2
0, ha s1 = s2
pozitív, ha s1 > s2

```
int strncmp(char* s1, char*s2, int maxhossz)
```
Mint az `strcmp`, de legfeljebb `maxhossz` karaktert hasonlít össze.

```
char* strcpy (char* cel, chsr* forras)
```
A `forras` stringet a `cel` stringbe másolja, beleértve a záró 0-át is. Visszatérési értéke a `cel` string mutatója.

```
char* strncpy (char* cel, chsr* forras, int maxhossz)
```
Mint az `strcpy`, de legfeljebb `maxhossz` karaktert másol.

21.2.3 Konvertáló függvények

Az `stdlib.h` header file kell!

```
int atoi(char* s)
```
Az s stringet egész számmá konvertálja. A konverzió megáll az első nem megfelelő karakternél.

```
long atol(char* s)
```
Az s stringet hosszú egész számmá konvertálja. A konverzió megáll az első nem megfelelő karakternél.

```
double atof(char* s)
```
Az s stringet lebegőpontos számmá konvertálja. A konverzió megáll az első nem megfelelő karakternél.

21.3. File kezelő függvények

A C alapvetően byte-ok (pontosabban `char` típusú egységek) sorozataként tekinti a file-okat. A programozó feladata a file tartalmát értelmezni. Egyetlen kivétel: ha a file-t ***text*** típusúnak deklaráljuk, a `'\n'` ("sorvég") karakter operációs rendszer függő kezelését elfedi előlünk.

21.3.1 File megnyitása és lezárása

Minden file-t egy `FILE` előredefiniált típusú struktúra azonosít. Erre mutató pointert (`FILE*` típusút) használnak a file-kezelő föggvények. Van három előre definiált `FILE*` változó:

`stdin`	: standard input
`stdout`	: standard output
`stderr`	: standard hibacsatorna

Minden file-t a használata előtt meg kell nyitni. Ezzel kapunk egy egyedi azonosítót, amellyel a későbbiekben hivatkozunk rá, illetve beállíthatjuk bizonyos jellemzőit. Erre szolgál az `fopen` függvény:

```
FILE* fopen(char* filenev, char* mod)
```

ahol
`filenév` paraméter a file neve (az adott operációs rendszer szabályai szerint)

`mod` egy, két vagy három karakteres string, amely meghatározza a file használatának a módját, az alábbiak szerint:

Mód jele	Használat módja	A megnevezett file-lal való kapcsolat
r	Csak olvasásra	Léteznie kell!
w	Csak írásra	Ha létezik, előző tartalma elvész, ha nem létezik, létrejön egy új (üres) file.
a	Hozzáírás	Ha létezik, az írás az előző tartalom végétől kezdődik. Ha nem létezik, létrejön egy új file.
r+	Olvasás és írás	Léteznie kell!
w+	Olvasás és írás	Ha létezik, előző tartalma elvész, ha nem létezik, létrejön egy új (üres) file.
a+	Olvasás és hozzáírás	Ha létezik, az írás az előző tartalom végétől kezdődik. Ha nem létezik, létrejön egy új file.

A fenti hat mód jel bármelyike után írható a **b** vagy **t** karakter, jelezve a bináris vagy text (szövegfile) jelleget. A különbség csak az, hogy a bináris filet mindig byte-onként kezeli a program, míg szövegfile esetén a '\n' karakter kiírása az operációs rendszer által sorvégjelként értelmezett byte-sorozatot szúr be a file-ba, az ilyen byte-sorozat olvasása pedig egy '\n' karaktert eredményez.

Példák:
`"a+t"` szövegfile megnyitása hozzáfűzésre
`"rb"` bináris file megnyitása csak olvasásra.

A függvény visszatérési értéke a file leíró struktúrára mutató pointer, ha a megnyitás sikeres, a `NULL` pointer egyébként. Ezért szokásos használata:

```
FILE* inputfile;

  .  .  .

if ( (inputfile=fopen("bemeno.txt","rt")) == NULL)
  {
  fprintf (stderr,"File nyitási hiba az input file-nál!\n");
  exit(1);       /* A program futásának befejezése */
  }
else
  {
  /* A file feldolgozása */
  }
```

Minden file-t a használata után le kell zárni. Erre szolgál az

```
int fclose(FILE* f)
```

függvény, amelynek paramétere a lezárandó file leírója, visszatérési értéke 0 sikeres lezárás esetén, `EOF` egyébként.

Megjegyzések:

- A file lezárása után az azonosítására használt változó újra felhasználható.
- A főprogram (main függvény) végén minden nyitott file automatikusan lezáródik, mielőtt a program befejezi a futását.
- A program nem közvetlenül a file-ba (többnyire a lemezre) ír, hanem egy bufferbe. A bufferből a file-ba az adatok automatikusan kerülnek be, ha a buffer megtelik. A file lezárása előtt a nem üres buffer is kiíródik a file-ba. Ha azonban a program futása rendellenesen áll le, nem hajtódik végre automatikusan file lezárás, így adatok veszhetnek el.

21.3.2 Írás és olvasás

```
int fprintf(FILE* f, char*s, . . . )
```
Mint a `printf`, csak az `f` file-ba ír ki.

```
int fscanf(FILE* f, char*s, . . . )
```
Mint a `scanf`, csak az `f` file-ból olvas.

```
int getc(FILE* f)
```
Mint a `getchar`, csak az `f` file-ból olvas.

```
int putc(char c, FILE* f)
```
Kiírja a `c` karaktert az `f` file-ba. Visszatérési értéke a kiírt karakter, hiba esetén `EOF`.

```
int fputs(char*s, FILE* f)
```
Kiírja az `s` stringet, a záró nulla karakter nélkül. Visszatérési értéke az utoljára kiírt karakter, vagy `EOF`.

```
char* fgets(char*s, int n, FILE* f)
```
Beolvas az `s` stringbe a sorvége karakterig, de maximum `n-1` karaktert. A string végére teszi a záró nulla karaktert. Visszatérési értéke a beolvasott stringre mutató pointer, vagy `NULL`.

Megjegyzés:

A fenti függvények elsősorban szöveges file-ok kezelésére szolgálnak, bár a `getc` és `putc` mindkét típus esetén használható, de sorvég-karakter esetén a működésük a file típusától függ.

```
int fread(void*p, int meret, int n, FILE* f)
```
A függvény `n` db, `meret` méretű adatot olvas be, és elhelyezi azokat a `p` címtől kezdődően. A `p` tetszőleges objektumot megcímezhet. A programozó felelőssége, hogy a kijelölt memóriaterület elegendő hosszúságú-e és a feltöltött memóriaterület értelmezhető-e. Visszatérési értéke a beolvasott adatok száma. Hiba vagy file vége esetén ez `n`-től kisebb.

```
int fwrite(void*p, int meret, int n, FILE* f)
```
Mint az `fread`, de kiír.

Megjegyzés:
A fenti függvények a bináris file-ok kezelésére szolgálnak.

21.3.3 Pozicionálás a file-ban

Minden file-hoz tartozik egy aktuális pozíció, amit byte-ban számítunk. Egy file megnyitása után az aktuális pozíció 0, azaz a file elejére mutat. A végrehajtott írási és olvasási műveletek az aktuális pozíciót is állítják, az átvitt byte-ok számának megfelelően. Az aktuális pozíciót a programból közvetlenül is állíthatjuk, amivel a file feldolgozása tetszőleges sorrendben megvalósítható.

```
int fseek(FILE* f, long offset, int viszonyitas)
```

Hozzáadja (előjelesen!) az `offset` értékét a `viszonyitas` által meghatározott értékhez, és ez lesz az új aktuális pozíció. `viszonyitas` lehetséges értékei (előredefiniált konstansok):

`SEEK_SET`	a file eleje
`SEEK_CUR`	az aktuális file pozíció
`SEEK_END`	a file vége

Visszatérési értéke 0 sikeres végrehajtás esetén, nem 0 egyébként.

```
long ftell(FILE* f)
```

Visszadja az aktuális file pozíció értékét, a file elejétől számolva. Sikertelenség esetén a visszatérési érték -1.

```
void rewind(FILE* f)
```

A file elejére állítja a file pozíciót.

Megjegyzés:
A file pozíció állítása a buffer kiírását is jelenti, ha az új pozíció nem a bufferen belül van.

21.4. Példaprogram: string keresése file-ban

Írjunk programot, amely az első paraméterében megadott stringet megkeresi a második paraméterében megadott file-ban. Ha a harmadik paraméter "p", akkor pontos (kis/nagybetű érzékeny) egyezést keres, ha elmarad, vagy bármi más, akkor csak "betűegyezést" keres. A program csak akkor kezdje el a működését, ha a paraméterek száma megfelelő. Jelezze ki, ha a megadott file megnyitása nem sikerült. A visszaadott státuszkód jelezze a hiba okát!
A program minden egyezésnél írja ki a sor és a soron belül az első egyező karakter számát!

/* KERES.C */

```
/* Készítette: Ficsor Lajos */

/* A program megkeresi egy szövegfile-ban egy string
   előfordulásait
*/

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

#define MAXSORHOSSZ 300

int fgetstrl (FILE* inp, char* s, int max);

FILE* inp;

int main(int argc, char* argv[])
{
int pontos;
int sorhossz;
char sor[MAXSORHOSSZ+1];
int mintahossz;
int sorszam = 0;
int i;

if (argc < 3 || argc > 4 )
  {
  fprintf(stderr, "\nHasznalata: keres minta filenev [p]\n");
  exit(1);
  }

if ( (inp=fopen(argv[2], "r")) == NULL)
  {
  fprintf(stderr, "\nFile nyitasi hiba!\n");
  exit(2);
  }

if (argc == 4 && argv[3][0] == 'p')
  {
  /* Pontos kereses kell! */
  pontos = 1;
  }
else
  {
  /* nem kell pontos kereses */
  pontos = 0;
  /* A minta nagybetusre konvertalasa */
  strupr(argv[1]);
  }

/* Minta hossza */
```

```
mintahossz = strlen(argv[1]);

/* Olvasas soronkent a file vegeig */
while ( (sorhossz=fgetstrl(inp, sor, MAXSORHOSSZ)) != EOF)
  {

  sorszam++;

  /* Ha nem kell pontos egyezes, nagybetusre konvertalas */
  if ( !pontos)
    {
    strupr(sor);
    }
  /* kereses a soron belul */
  for (i=0; i<=sorhossz-mintahossz; i++)
    {
      if ( strncmp(argv[1], sor+i, mintahossz) == 0)
      {
      /* Egyezes van! */
      printf("\n Sorszam: %d, karakter: %d", sorszam, i);
      }
    }

  } /* while */

fclose (inp);
return 0;
}

int fgetstrl (FILE* inp, char* s, int max)
/*****************************************************
  A fuggveny egy sort olvas be az inp file-bol,
  vagy annak az elso max karakteret, ha attol hosszabb,
  es az s tombbe helyezi el, string-kent.
  Visszateresi erteke a beolvasott karakterek szama,
  vagy EOF
*****************************************************/
{
int c;
int i;

i=0;
while ( (c=getc(inp)) != '\n' && c !=EOF && i<max)
  {
  s[i++] = c;
  }
s[i] = '\0';
return c==EOF ? c : i;
}
```

Ajánlott irodalom:

1. Brian W. Kernighan, Dennis M. Ritchie
   A C programozási nyelv
   Műszaki Könyvkiadó
   Budapest, 1988.

2. Brian W. Kernighan, Dennis M Ritchie
   A C programozási nyelv. Az ANSI szerint szerint szabványosított változat.
   Műszaki Könyvkiadó
   Budapest, 1997.

3. Benkő Tiborné, Poppe András, Benkő László
   Bevezetés a BORLAND C++ programozásba
   ComputerBooks
   Budapest, 1995

Miskolci Egyetem Általános Informatikai Tanszék kódolási szabványa C nyelvhez

Tartalom:

---

1. Bevezetés

1.1 A szabvány célja

A szabvány célja az egységes és helyes programozási stílus elsajátításának elõsegítése. A szabványnak megfelelõ programok egyszerûen áttekinthetõek, ebbõl következõen jól érthetõek, és könnyen karbantarthatóak lesznek. Az egyes kódrészek újrahasznosíthatósága megnövekszik, ezáltal a feladatok elkészítéséhez szükséges idõráfordítás is csökken.

1.2 Hatáskör

Az Általános Informatikai Tanszék számára készült valamennyi C nyelvû program kötelezõen meg kell felelnjen ennek a szabványnak. A szabvány formai elõírásai igazodnak a GNU kódolási szabványához, így a forrásfájlok formázásához a **GNU indent**, vagy a **GNU Emacs** program használata javasolt.

---

2. Fájlok

2.1 A forrás tagolása

Egy forrásfájlba csak logikailag szorosan összetartozó függvények kerülhetnek.
Össze nem tartozó függvények egy fájlban tárolásakor az egyes függvényeket nehezen lehet megtalálni, valamint újrafelhasználáskor egy függvény miatt a teljes object fájl összes függvénye hozzálinkelõdik a programhoz, ezzel fölöslegesen növelve a futtatható fájl méretét.

A forrásfájlok mérete nem lépheti túl az 500 sort.
Ez a szabály is az áttekinthetõséget szolgálja. Amennyiben egy 500 sor összterjedelmet meghaladó függvénycsoport nagyon szorosan kapcsolódik egymáshoz, azt az egyes forrás fájlok elnevezésével kell jelezni.

2.2 A fájlok elnevezése

A C forrás fájlok kiterjesztése mindig ".c", a header fájlok kiterjesztése mindig ".h".
A fájlnévbõl elsõ látásra megállapíthatjuk a fájl rendeltetését. Az egységes elnevezésbõl adódóan a fájlok kezelése is egységes lehet, egyszerûen készíthetünk fájlkezelõ eszközöket.
Az összetartozó C és header fájlok neve a kiterjesztéstõl eltekintve azonos.
Amennyiben több, szorosan összetartozó C fájlhoz egy header fájl tartozik, akkor a C fájlok nevének elsõ része megegyezik a header fájl nevével, második része pedíg egy szám, vagy a funkcióra utaló rövidítés.

Például:	`screen.h`	`screen1.c`	`screen2.c`
Esetleg:	`screen.h`	`screen_base.c`	`screen_complex.c`

A nevek egyezõsége elõsegíti a gyors, magától értetõdõ összerendelést, így növelve az áttekinthetõséget, és könnyítve az újrahasznosíthatóságot.
A fájlnevek legyenek egyediek minnél nagyobb környezetben, és utaljanak a tartalmazott függvény(ek) funkciójára.
Egy nagyobb projektben, - ahol több katalógusba kell szervezni a fájlokat, - vagy újra felhasználáskor zavaró lehet több, azonos nevû fájl megléte -, mégha különbözõ katalógusban találhatóak is. A funkcióra nem utaló fájlnév már néhány forrás fájl esetén is szinte lehetetenné teszi egy adott függvény megtalálását, ami nagyban megnehezíti mind a hibakeresést, mind a karbantartást.

2.3 Include fájlok

Header fájlokon kívül más fájl include-álása tilos !
Kezdõ programozóknál elõfordul az a nagyon rossz gyakorlat, hogy a több forrás fájlból álló programnál egyszerûen a C fájlokat include-álják a main függvényt tartalmazó forrás fájlba. Ez elõreláthatatlan hibákhoz, és óriási kavarodáshoz vezethet. A fordítási idõ is nagyban megnövekszik, mert minden fordítás alkalmával újrafordul az összes modul, nem csak a függõségekbõl eredõ szükségesek.
Minden header fájlban léteznie kell egy többszörös include-álást gátló mechanizmusnak.
Ha egy header fájl több, más header fájlba is include-álva van, akkor könnyen elõállhat az a helyzet, hogy egy header fájl többször kerül include-álásra ugyanabba a forrás fájlba. Ez fordítási hibákhoz (konstans újradefiniálás) vezethet, valamint növeli a fordítási idõt. Ezt a legegyszerûbben az **#ifndef/#define/#endif** szerkezettel akadályozhatjuk meg.

Példa:
```
#ifndef _SPECIALNAME_
#define _SPECIALNAME_

/* header block */

#endif
```

A _SPECIALNAME_ ajánlott képzési módja: a **filenev.h** include file-hoz használjuk az _FILENEV_H_ konvenciót.

Csak a minimálisan szükséges header fájlokat include-áljuk.
Ezzel biztosítható a minimális fordítási idõ, és a névazonosságból eredõ összeférhetelenség esélyét is a minimálisra csökkentjük.
Az elõre definiált header fájlokat a #include <filenév.h>, a felhasználói header fájlokat pedíg a #include "fájlnév.h" direktívával include-áljuk.
Ha a **#include** direktívát <> pár követi, akkor a fordító a header fájlt a megadott include katalógusokban keresi, ezzel szemben ha a fájlnév "" jelek között szerepel, akkor a fordító csak az aktuális katalogógusban keresi a header fájlt. Így csökken a fájlnév ütközés veszélye is.

2.4 Kommentek

Minden forrás fájl elején szerepelnie kell az adott fájlra vonatkozó általános információkat tartalmazó megjegyzés blokknak.
A blokk kinézete:

```
/*
        fájlnév

A fájlban lévõ függvények együttes funkciója
A készítõ neve (esetleg Copyright)
Utolsó módosítás dátuma, verziószám
*/
```

Minden függvény deklarációja és definíciója elõtt részletes megjegyzésnek kell szerepelnie.
A header fájlban a függvény használójának szóló, a C forrás fájlban pedíg a program karbantartójának, esetleges továbbfejlesztõjének szóló megjegyzést kell elhelyezni. A megjegyzés blokk részletes ismertetése a függvények-rõl szóló szabályoknál található.
A megjegyzések egy fájlon -, lehetõleg az egész projekten - belül azonos nyelven, vagy csak magyarul, vagy csak angolul legyenek írva.
Zavaró lehet több nyelv keverése, esetleg félreértést is okozhat, az olvashatóságot mindenképpen rontja. Feladatok elkészítése során magyar nyelvû (de az eltérõ kódtáblák miatti problémákat elkerülendõ természetesen *ékezet néküli)* megjegyzéseket használjunk.

---

3. Nevek

A név mindig utaljon az általa azonosított elem funkciójára.
Akár függvényrõl, konstansról, vagy változóról legyen is szó, csak így lesz egyszerûen érthetõ és átlátható a program. Ellenkezõ esetben minden elõforduláskor meg kell keresni a definíció, vagy az utolsó hivatkozás helyét, hogy értelmezhessük a név jelentését. A jó névválasztás egyszerûsíti, egyes esetekben fölöslegessé is teheti a megjegyzéseket.
A makrók (#define) neve mindig csupa nagybetûbõl álljon.
Ez által a makrók elsõ látásra elkülönülnek a programszöveg többi részétõl. Erre azért van szükség, mert a makrók - fõként a függvények - kezelése elõvigyázatosságot igényel. Lásd a konstansok és a függvények fejezeteket.
A több szóból álló neveket egybe kell írni, és minden szót nagybetûvel kell kezdeni.
Több szóban pontosabban lehet utalni az azonosított elem funkciójára. A kis- és nagybetûk megfelelõ alkalmazásával a szóhatárok egyértelmûek. A makrók nevében a szóhatárokat az '_' karakterrel kell jelölni.

```
Például:        NotTooLongName
                THIS_IS_A_MACRO
```

Egy név láthatósága minnél nagyobb, annál hosszabbnak kell lennie.
A hosszú név csökkenti a névütközés lehetõségét. Egy könyvtár megírásakor minden név elé a

könyvtárra utaló prefixet kell alkalmazni. A prefix és a név közé '_'-t kell tenni. A prefix mindig kezdõdjön kis betûvel, kivéve a makróneveket.

```
Például:        disc_FunctionName
                DISC_MACRO_NAME
```

Sose használjunk egy vagy két aláhúzással (_ vagy __) kezdõdõ neveket

A fordítók legtöbbje mûködés közben ilyen formájú neveket használ belsõ azonosításra, ezért esetleg felderíthetetlen fordítási hibát eredményezhet alkalmazásuk.

Egyetlen kivétel lehet: a header fájlok többszöri include-álását megakadályozó mechanizmus. Az itt megadott névnek minden más névtõl különbözõnek, a header fájl funkciójára utalónak, és hosszúnak kell lennie.

Sose használjunk olyan neveket, amelyek csak a kis- és nagybetûs írásmódban, vagy az aláhúzás meglétében, ill. hiányában térnek el.

Ebben az esetben egy elírás esetleg nem okoz szintaktikai hibát, ezért a fordító nem figyelmeztet rá. Az ebbõl eredõ szemantikai hibák felderítése különösen nehéz. Egyebként is nehez fejben tartani, hogy melyik írásmód melyik elemet azonosítja.

```
Ilyet ne:       ThisIsaName
                THISisaName
                ThisIs_A_Name
```

A nevekben ne alkalmazzunk félreérthetõ rövidítéseket.

Ami a program írásakor a program írójának egyértelmû, az nem biztos, hogy késõbb, egy másik személynek is az lesz.

```
Például:        TermCap         /* Terminate Capture or
                                   Terminal Capability ? */
```

---

4. Függvények

Egy függvény hossza nem haladhatja meg a 100-150 sort.

Ettõl hosszabb függvények nehezen áttekinthetõek, nehezen értelmezhetõek, ezért a hibakeresés is nehéz. Ha egy hosszú függvény futását valamilyen hibaesemény miatt meg kell szakítani, akkor bonyolult a függvény elõzõ tevékenységeit visszaállítani (undo).

A 100-150 soros terjedelem csak abban az esetben léphetõ át, ha a tevékenység olyan, szorosan összefüggõ utasításokat tartalmaz, amelyeket nem lehet logikusan szétbontani új függvényekre, és sok (5-nél több) paraméter átadása lenne szükséges. További feltétel, hogy az ilyen hosszú függvények nem tartalmazhatnak 3 szintnél mélyebben egymásba ágyazott elágazásokat. A nagy elágazási mélység nehezen tesztelhetõvé teszi a függvényt.

4.1 Argumentumok

Törekedjük minnél kevesebb argumentum használatára.

A sok argumentum rossz tervezésre utal. Sok argumentum esetén nehéz fejbentartani az argumentumok sorrendjét, azonos típusú argumentumok felcserélése esetén nehezen felderíthetõ szemantikai hiba keletkezik.

Tömböket és (nagy méretû) struktúrákat mindig mutatón keresztül adjunk át.

Az argumentumok függvényhíváskor átmásolódnak a stack-re. Nagy méretû tömbök, vagy struktúrák másolása hosszú ideig tart, rontja a kód hatásfokát. Bizonyos számítógép architektúráknál a stack mérete viszonylag kicsire van korlátozva. Ezért akár egyetlen, nagyobb méretû tömb átadása is stack-túlcsordulást okozhat.

A függvény prototípusok megadásánál az argumentumok típusai mellett mindig adjuk meg azok neveit is.

A nevek megadása az ANSI-C szabvány szerint nem kötelezõ, de a funckióra utaló nevek

alkalmazásával a függvény használata egyszerûbben megjegyezhetõ, a függvény egyszerûbben dokumentálható.

Kerüljük a meghatározatlan (...) paraméter-átadást.

A meghatározatlan paraméter-átadásra a legismertebb példa a **printf()** függvény. A második és további argumentumok száma és típusa meghatározatlan. Ebbõl fakadóan a fordító nem tudja ellenõrizni a függvényhívás helyességét, szemantikai hibák elkövetésére nagyobb a lehetõség. Ezért csak végsõ esetben alkalmazzunk meghatározatlan paraméter-átadást.

4.2 Visszatérési érték

Mindíg adjuk meg a függvény visszatérési értékének típusát.

A típus hiánya a programot értelmezõ számára zavaró lehet, mert nem azt jelenti, hogy nincs visszatérési érték (**void** típus), hanem a fordító alapértelmezése szerinti **int** értéket jelent.

Tömböket és struktúrákat mindig mutatón keresztül adjuk vissza.

Az argumentumok átadásához hasonlóan itt is másolás történik, ami nagy mennyiségû adat esetén sokáig tart.

Sose adjuk vissza valamelyik lokális változó címét.

A függvény visszatérésekor elõször felszabadul a stack-en a lokális változók által elfoglalt terület, a vezérlés ezután kerül vissza a hívó függvényhez. A visszakapott mutató pedíg egy érvénytelen (felszabadított) területre mutat. Az elsõ mutatómüvelet **szegmentációs hibát** fog okozni. (DOS alatt nem kapunk hibajelzést, csak valamilyen, esetleg véletlenszerû értéket, és ez még rosszabb.)

Ha a függvényen belül dinamikusan foglalunk memóriát a visszatérési értéknek, akkor ezt egyértelmûen dokumentálni kell.

Ilyen esetekben ugyanis a hívó függvényben kell gondoskodni a memóriaterület felszabadításáról, amit sok programozó elfelejt. Az eredmény: futás közben fokozatosan növekvõ memóriaszükséglet, esetleg instabil program.

4.3 Kommentek

A nagyobb lélegzetõ, több soros megjegyzéseket az elsõ oszlopban kell kezdeni, és a magyarázott blokk elõtt kell elhelyezni.

Az ilyen típusú megjegyzések általában egy nagyobb programrészlet, esetleg egy egész függvény mûködését, vagy funkcióját írják le. A programszöveg jobb olvashatósága miatt a megjegyzésnek el kell különülnie a többi szövegtõl. Minden függvény definíciójánál kötelezõ egy ilyen megjegyzésblokk alkalmazása a következõ formában:

```
/*
Feladata:
Használatának elõfeltételei:
Bemenõ paraméterek:
Kimenõ paraméterek (visszatérési érték):
Egyéb paraméterek:
Hivatkozott globális változók:
Hívott függvények:
Készítõ:
Utolsó módosítás dátuma:
*/
```

Az egyes sorok megléte akkor is kötelezõ, ha adott esetben valamelyikhez nem tartozik semmi.
A használat elõfeltételeihez tartozik minden olyan elõzetes tevékenység, ami a függvény megfelelõ mûködéséhez szükséges. Pl: egy fájl megnyitása, bizonyos beállítások, vagy valamilyen adatstruktúra feltöltése.
A bemenõ paramétereknél le kell írni az összes bemenõ argumentum funkcióját, lehetséges

értéktartományát, az egyes argumentumok összefüggéseit.
A kimenõ paraméterek között a visszatérési értéken kívül az összes olyan argumentumot -, és azok lehetséges értéktartományát - is fel kell tüntetni, amelyeket a függvény megváltoztathat.
A hivatkozott globális változóknál meg kell adni, hogy mely változók értékét változtatja meg a függvény, és az egyes változóknál mi az elvárt bemenõ, ill. kimenõ értéktartomány.
A hívott függvények listáján csak a nem szabványos függvényeket kell felsorolni. Ez a fordítási függõségek könnyebb meghatározásánál hasznos.
A rövid, egysoros megjegyzéseket a magyarázott utasítással egy sorba, vagy az utasítást követõ sorba kell elhelyezni úgy, hogy az utasítástól jól elkülönüljön.
Ezek a megjegyzések egy fontos, esetleg bonyolult utasítást magyaráznak. Amennyiben az utasítás és a magyarázat olyan rövid, hogy elférnek egy sorban, akkor minimum 8-10 helyközt kell köztük hagyni. Ha az utasítás vagy a megjegyzés hossza nem teszi lehetõvé az egy sorba írást, akkor a megjegyzést a következõ sorban kell kezdeni egy plussz bekezdésnyivel az utasítástól beljebb. Ilyenkor a megjegyzés után üres sort kell hagyni, és a következõ utasítás csak az üres sor után következhet. A megjegyzésnek mindig jól el kell különülnie az utasításoktól, különben rontja az olvashatóságot, és az ellenkezõ hatást éri el, mint amire szánták. A megjegyzésnek azokat a dolgokat kell magyarázni, amelyek nem látszanak, pl. mi a jelentése egy feltételnek, mikor, vagy mitõl következhet be. A triviális dolgok kommentezése hiba.
Példa:
```
        FontosFuggveny( ElsoParameter, MasodikParameter );
          /* Ez itt a hozzatartozo megjegyzes */

        Utasitas();          /* Rovid megjegyzes */
```

4.4 Írásmód

A függvény visszatérési értékének típusát a prototípust (vagy fejlécet) megelõzõ sorba kell írni.
Így a függvény neve kerülhet a sor elejére, a függvényt könnyebb lesz megtalálni.
Ha a függvénynek olyan sok argumentuma van, hogy nem férnek ki egy sorba, akkor minden argumentum nevét a típusával együtt egy bekezdéssel új sorba kell írni.
A paraméterek így jól elkülönülnek egymástól, a sorrendjük is jól követhetõ.
Példa:
```
        int
        Fuggvenynev( int ElsoParameter,
          char *MasodikParameter,
          float HarmadikParameter );
```

---

5. Konstansok

A programszövegben mindig szimbólikus konstansokat használjunk.
Ennek egyik oka, hogy a program értelmezése nagyon nehézkessé, szélsõ esetben szinte lehetetlenné válik. Minden egyes elõfordulást megjegyzéssel kellene ellátni, s mint láttuk, a sok komment szintén rontja az olvashatóságot.
A másik ok, hogy ha egy konstans többször is elõfordul egy programban, és késõbb meg kell változtatni az értékét, akkor végig kell bogarászni a teljes forrásszöveget, - esetleg több fájlt is -, hogy megtaláljuk és kicseréljük az összes elõfordulást. Ha csak egy-két helyen elfelejtjük átírni a konstans értéket, az "katasztrófához" is vezethet. Az egyszerû, automatikus keres-cserél nem alkalmas, mert lehet, hogy szerepel ugyanilyen érték más funkcióval. Szimbólikus konstansok esetén az értéket csak egy helyen - a konstans definíciónál - kell megváltoztatni.
A szabály alól csak az egyedi szövegkonstansok és a 0, 1, -1 számértékek jelenthetnek kivételt. A szövegkonstansok nagy része csak egyszer fordul elõ a forrásszövegben, és viszonylag hosszú,

ezekre fölösleges szimbólummal hivatkozni. A 0, 1, -1 értékek jelentése pedíg általában világos, ha mégsem, akkor ezeket is lehet szimbólikusan megadni, vagy megjegyzéseket alkalmazni.

A konstansokat a #define helyett lehetõleg a const vagy az enum használatával definiáljuk.

A makrókat fordításkor szövegbehelyettesítéssel értelmezi az elõfordító. Ezért a lefordított bináris fájlban már semmi sem utal a konstans nevére. A legtöbb debugger nem tudja megjeleníteni az ilyen konstansok nevét, csak az értéke fog szerepelni a megfelelõ helyeken. Ha egyszerre több, összetartozó konstanst szeretnénk definiálni, akkor használjuk az enum konstrukciót. Ha mégis a **#define**-t használjuk, akkor ügyeljünk arra, hogy a negatív számokat mindig tegyük zárójelek közé, ezzel csökkenthetjük a mellékhatások számát és valószínûségét.

Például (ilyet ne):

```
                #define HETFO    1
                #define KEDD     2
                .
                .
                #define VASARNAP 7

                #define SZORZO -3
```

Inkább:

```
                enum NAPOK { HETFO = 1, KEDD, SZERDA, CSUTORTOK, PENTEK,
                  SZOMBAT, VASARNAP };

                #define SZORZO (-3)
```

Sose használjunk olyan makrófüggvényt, amelyben az argumentumokra egynél több hivatkozás van.

A makrófüggvényeket nagyon óvatosan kell kezelni, mert úgynevezett mellékhatások léphetnek fel, és esetleg a függvény sem azt az eredményt szolgáltatja, amit elvárnánk tõle.

Például:

```
        #include<stdio.h>

        #define NEGYZET( a ) ( (a) * (a) )

        void main( void )
        {
          int szam = 2;

          printf( "szam=%d, ", szam );
            /* szam = 2 */

          printf( "NEGYZET( ++szam )=%d, ", NEGYZET( ++szam ) );
            /* ++2 = 3, NEGYZET( 3 ) = 9, NEGYZET( ++szam ) = 16 !!! */

          printf( "szam=%d\n", szam );
            /* szam = 4 */
        }
```

---

6. Változók

A változókat a lehetõ legkisebb hatáskörrel deklaráljuk.

Ez növeli a kód olvashatóságát, csökkenti a névütközés lehetõségét, és csökkenti a program memóriaigényét, bizonyos esetekben még a futási sebességet is javítja. Ha egy változó deklarációja és használata egymástól távol esik, akkor nehéz a változó típusának megállapítása, elõre-hátra kell lapozni a szövegben.

A névütközés nem okoz szintaktikai hibát, mert a kisebb hatáskörû változó az adott blokkban elfedi a nagyobb hatáskörû változót. Ebbõl szemantikai hiba adódhat, félreértést okozhat.

Ha egy változót egy függvény elején deklarálunk, és csak egy feltételes blokkban használunk, akkor a változónak szükséges hely abban az esetben is lefoglalódik, majd felszabadítódik, ha a blokk

végre sem hajtódik. A program futása gyorsítható azáltal, ha a változót csak a feltételes blokk elején deklaráljuk.

Minden változót külön sorban deklaráljunk.

Az olvashatóság jobb lesz, az adott változó funkciójára utaló megjegyzés a deklarációval egy sorba írható. Ez vonatkozik a struktúrák és az uniók mezõinek deklarálására is.

Mutatók használatánál ügyelni kell, hogy az elsõ értékadás mindig elõzze meg az elsõ hivatkozást.

A mutatók a többi változóhoz hasonlóan deklarációkor 0-ra vagy sehogyan sem inicializálódnak. Amikor egy ilyen mutatóra hivatkozunk, biztos a **Segmentation Fault**, vagy valamilyen hasonló futási hibaüzenet. Bõvebben lásd a Memória kezelés fejezetet.

---

7. Vezérlési szerkezetek

A feltételes blokkok, ciklusmagok írásmódja:

A blokkot nyitó kapcsos zárójelet '{' a feltételes utasítással egy oszlopba, de az azt közvetlenül követõ sorba kell írni. A blokkot záró kapcsos zárójel '}' szintén ebben az oszlopba kell hogy kerüljön. A blokkban szereplõ összes utasítás egy bekezdéssel (2, 4 karakter, de mindig azonos) beljebb kerül. Ha egy utasítás nem fér ki egy sorba, akkor a következõ sorban még egy bekezdéssel beljebb kell folytatni. Amennyiben még ez sem elég, akkor a következõ sorokban már nem kell újabb bekezdést tenni. Ez alól csak a hosszú szövegkonstansok jelentenek kivételt, ahol a bekezdés helyköz karakterei is a szöveg részét képeznék. Minden több sorba írt utasítás után célszerû egy üres sort hagyni, hogy a következõ utasítás jól elkülönüljön.

Például:

```
        if ( feltetel )
        {
          EzEgySokParameteresFuggveny( ElsoParameter,
            MasodikParameter,
            HarmadikParameter );

          printf( "Ez itt egy nagyon hosszú szövegkonstans, amit\
         muszály több sorba írni, ezért a sorok elején csak egy\
         helyköz hagyható.\n" );
        }
        else
        {
          Ut1;
          .
          .
        }
```

Ha egy blokk hosszabb, mint egy képernyõoldal, vagy több blokk van egymásba ágyazva úgy, hogy a blokkok vége nagyon közel van egymáshoz, akkor a záró }-hez megjegyzést kell írni.

A bekezdésekbõl ugyan látszani kell, hogy melyik } melyik blokk végét jelzi, de hosszú blokkok esetén sokat kell elõre-hátra lapozni a szövegben, hogy megtaláljuk a párokat. A megjegyzésben feltüntetve a feltételt, vagy a ciklus utasítás elsõ sorát sokat segíthet a megértésben, és jobbá teszi az áttekinthetõséget. Ez a szabály vonatkozik az elõfordító direktívák használatára is. (#ifndef ... #endif)

Például:

```
        switch( szelektor )
        {
          case FEL:
          {
            Utasitasok;
            break;
          }  /* FEL */
```

```
          case LE:
          {
            Utasitasok;
            break;
          }  /* LE */

          default:
          {
            Hiba;
            break;
          }  /* default */
        }  /* switch( szelektor ) */
vagy
        for( index = 0; index < MERET; index++ )
        {
          Ciklusmag;
        }  /* for( index = 0; index < MERET; index++ ) */
```

A switch kifejezésben mindig legyen default ág a váratlan esetek kezelésére. Ennek az ágnak elegendõ valamilyen egyszerû hibaüzenetet tartalmaznia, esetleg exit utasítással kilépni a programból. Ez még mindig jobb, mintha a program zavartalanul fut tovább valamilyen jelzés nélküli szemantikai hibával. **A case ágak végénél mindig gondoljunk végig, szükséges-e a break utasítás!** A **break** hiányában a program futása a kövtkezõ **case** ágon folytatódik mindaddig, amíg az elsõ **break**-hez, **return**-höz, vagy a **switch** kifejezés végére nem ér a program futása. A program látszólag hiba nélkül fut, mégis szemantikailag értelmetlen eredményeket szolgáltathat. Nehezen kideríthetõ hiba.

A goto utasítás használata szigorúan TILOS !!!

A **goto** megtöri a program futását és az elõreláthatatlan helyen fut tovább. A program futása követhetetlenné válik. Másrészt a C nyelv olyan változatos ciklusokat, vezérlési szerkezeteket tesz lehetõvé, hogy nincs is szükség a **goto** használatára. A **continue**, **break** utasításokkal a ciklusok és feltételes blokkok végrehajtása könnyedén megszakítható, a futás mégis áttekinthetõ marad, mert az mindig a blokkban elsõ, illetve a blokkot követõ elsõ utasításon folytatódik.

A ciklusokból való kilépésre flag-ek helyett a break utasítást használjuk.

Ezáltal egyszerûsödik a ciklus végrehajtási feltétele, egy változóval kevesebbet kell használni, és a program is áttekinthetõbb lesz.

Például (ilyet ne):

```
        int flag = 0;

        for( index = 0; index < HATAR && !flag; index++ )
        {
          Utasitasok1;
          if ( feltetel ) flag = 1;
          Utasitasok2;
        }

Inkább:
        for( index = 0; index < HATAR; index++ )
        {
          Utasitasok;
          if ( feltetel ) break;
        }
```

Feltételek megadásakor (ha csak az algoritmus nem ír elõ kifejezetten mást) használjunk alul zárt, felül nyitott halmazokat.

Az **x>=20 és x<=50** helyett használjuk a **x>=20 és x<51** formát. Ennek több elõnye is van:

- Az intervallum mérete megegyezik a határok különbségével.
- A határok egyenlõek üres intervallum esetén.
- A felsõ határ sosem kisebb az alsó határnál.

8. Kifejezések

Helyközök használata:
Általános elv, hogy mindig az áttekinthetõséget és az olvashatóságot kell szemelõtt tartani.
Mindíg helyközt kell tenni:

- az unáris (egyoperandusú) opertátorok ('.', '->', '++', '--', ...) kivételével minden operátor elé és mögé;
- a vesszõ után;
- a pontosvesszõ és a magyarázat közzé;
- a ()-be és []-be zárt kifejezés elé és mögé kivéve, ha az üres, vagy csak egy tagú;
- az **if** utasítás és a nyitó zárójel közé.

Sosem szabad helyzözt tenni:

- vesszõ és pontosvesszõ elé;
- az unáris operátorok és az operandusuk közé;
- a ciklus utasítás, vagy a függvény neve és a nyitó zárójel közé;
- a tömb neve és a nyitó szögletes zárójel közé.

Ha egy kifejezés kiértékelési sorrendje nem triviális, akkor használjunk zárójeleket, még abban az esetben is, ha a fordító szempontjából fölöslegesek.
Kevésbé gyakorlott programozóknak nehézséget jelenthet egy összetett kifejezés megfelelõ értelmezése. A zárójelek használatával ez egyszerûsíthetõ. Másrészt, a fordító nem mindent a matematikában megszokott értelmezés szerint fordít le. Ezeket az eseteket is egyértelmûvé lehet tenni zárójelek alkalmazásával.
Például:

```
        a < b < c       /* ( a < b ) < c és nem ( a < b ) && ( b < c ) */

        a & b < 8       /* a & ( b < 8 ) és nem ( a & b ) < 8 */

        a++ - ++a--     /* Mit is jelent ez ??? */
```

Egy hosszú kifejezés több sorba is írható, ilyenkor az új sort mindig egy bekezdéssel beljebb kell kezdeni úgy, hogy az operátor (kivéve a záró zárójelet) a következõ sorba kerüljön.
Mindig a szöveg áttekinthetõségére kell törekedni. Az ilyen, többsoros kifejezések után egy üres sort kell hagyni, így az egyes kifejezések, illetve utasítások elkülönülnek egymástól.

9. Memória kezelés

A lefoglalandó méret kiszámításánál mindig használjuk a sizeof makrót.
Akár egyedi változónak, akár tömbnek foglalunk helyet, mindenképpen elõnyös a **sizeof** makró használata. Ha struktúrának, vagy struktúratömbnek akarunk memóriát foglalni, és a fejlesztés során megváltozik annak mérete, - pl. új mezõk hozzáadása, - akkor sem kell a folgalalási részt módosítani, mert az automatikusan követni fogja a méretváltozást. Elemi típusok - pl. int - is hasznos a **sizeof** használata, mert így jobban hordozható lesz a program.
Egy adott blokk felszabadítása lehetõ legközelebb kerüljön a blokk lefoglalásához.
Ezáltal kisebb az esély, hogy a blokk felszabadítatlanul marad, és jobban nyomonkövethetõ a lefoglalás/felszabadítás menete.
Minden blokk lefoglalását ellenõrizzük.
A **malloc és rokonai** NULL-lal térnek vissza sikertelen foglalás esetén. Amennyiben ezt nem ellenõrizzük, a mutatóra való elsõ hivatkozás **Segmentation Fault** futási hibához fog vezetni. (DOS alatt ilyen nincs. Sajnos !) Ellenõrzött foglalással ez elkerülhetõ.

Sorozatos helyfoglalásnál minnél nagyobb blokkokat foglaljunk le.
Dinamikus tömbök, vagy láncolt listák kezelésénél - kivéve, ha egy elem mérete meghaladja a 100-150 bájtot - ne egyesével, hanem minimum 50-100 bájtos darabokban foglaljuk le a memóriát. Ennek az az oka, hogy az operációs rendszerek nem "szeretik" a sok, apró blokkot, lelassulhatnak a memóriamûveletek, és egyéb káros események történhetnek.
Egy blokk felszabadítása után a mutatójának adjunk NULL értéket.
Ezáltal a programban bárhol ellenõrizhetõvé válik, hogy a mutató használható területre mutat-e, vagy egy használaton kívüli mutató-e. Ennek szintén a **Segmentation Fault** hiba elkerülésénél van jelentõsége.
Amennyiben egy függvényben dinamikus memóriakezelés van, azt a függvény fejrészénél lévõ megjegyzésben egyértelmûen jelezni kell.
Így csökkenthetõ az esély arra, hogy a blokk felszabadítatlan marad.

---

10. Hordozhatóság

Semmilyen feltételezést ne tegyünk a típusok méretére, vagy elõjelére vonatkozóan.
Különbözõ fordítók, különbözõ operációs rendszerek más-más mérettel kezelik az egyes típusokat. Az ebbõl adódó hibák elkerülésére mindig használjuk a **sizeof** makrót, valamint a **long** és a **short** típusmódosítót.
Hasonló a helyzet az elõjelességgel is. A **char** típust bizonyos fordítók elõjelesnek, mások elõjel nélkülinek tekintik. Ezért 8 bites ASCII értéket tároló változók deklarációjánál mindig használjuk az **unsigned** típusmódosítót.
Amikor típuskonverzióra van szükség az explicit konverziót részesítsük elõnyben az implicittel szemben.
Ez különösen a numerikus típusoknál szívlelendõ meg, mert különbözõ architektúráknál esetleg nem az az automatikus konverzió következik be, mint amit feltételeztünk, és így hibás eredményt adhat a kifejezés.
Ne írjunk olyan kódot, amelynek a mûködése valamelyik változó túl- vagy alulcsordulásától függ.
Más-más architektúrákon más-más lehet egy adott típusú változóban tárolható maximális illetve minimális érték, ezért a túl- illetve alulcsordulás is más értéknél következhet be, ami a program hibás mûködéséhez vezethet. Nehezen kideríthetõ hiba!
Ne tételezzünk fel egy adott végrehajtási sorrendet egy kifejezésen belül.
A végrehajtási sorrendet az optimalizáló fordítók átrendezik. Mindegyik fordító másképpen. A tényleges sorrendet pedig nem tudjuk megállapítani. Ezekben az esetekben a zárójelezés segíthet.
Ne tételezzük fel, hogy a függvényhívásban az argumentumok a prototípusnál megadott sorrendben adódnak át.
Az elõzõ pontban mondottak itt is érvényesek. Az átadás sorrendje fordítófüggõ. Ez leginkább a mellékhatások kihasználásánál vezethet hibás mûködéshez. Ennek elkerülésére függvényhívásokban több, egymással közvetlen kapcsolatban lévõ argumentum esetén semmilyen mellékhatást eredményezõ kifejezést se használjunk.

A fenti dokumentum az

ELLEMTEL Telecommunication Systems Laboratories Sweden

Programming in C++
Rules and Recommendations

címû angol nyelvû dokumentum C nyelvre adaptálása és átdolgozása.
Az erdeti dokumentum írói: Mats Henricson és Erik Nyquist.
Angolra fordította: Joseph Supanich.
A C nyelvre adaptálást és a magyar nyelvre fordítást készítette: Straub Kornél.
Ellenõrizte: Ficsor Lajos
Utolsó módosítás: 1999. március 20.