A C programozási nyelv

1 Előszó

Ez a jegyzet a „Programozási alapfogalmak” című jegyzettel együtt teljes értékű. A fogalmak definíciói és magyarázata nagyobbrészt abban a jegyzetben találhatók.

2 Bevezetés

2.1 Története címszavakban:

1972-ben Dennis Ritchie (assemblyt helyettesítő nyelvként, operációs rendszer írására)

1973 Ritchie és Ken Thompson UNIX C nyelven

Kezdetben a UNIX-hoz kötődött, később más implementációk is megjelentek
Kezdetben PDP-11 gépen használták, később más gépeken is.

Már első törekvés a forrásszintű gépfüggetlenség. Ez magyarázata a terjedésének.

1983-ban az ANSI szabványosítani kezdte.
1988-tól ANSI C szabvány.

2.2 Jellemzése

- általános célú
- gépfüggetlen
- a periféria kezelés nem a nyelv része
- nem támogatja a párhuzamos feldolgozást (parallel C)
- gazdag függvény könyvtár

- nehéz a kezdőknek: nincs sok merev szabály, nagy a szabadság, az értelmetlen dolgokat is megengedi
- általános célú, de nem minden feladatra ideális, pl. mivel I/O lehetőségei szegényesek adatfeldolgozásra nehézkesebb, nagyszerűen használható sokféle gépközeli programozási feladatra (közvetlenül címezhető memóriapozíció és I/O portok)
- a PC-s verziók gazdagabbak I/O egységek kezelésében
- a C++ tovább fejlődése a C-nek (egyrészt OOP felé, másrészt az értelmetlen dolgok nem megengedése felé)

2.3 Egy helló világ példa program.

```
#include <stdio.h>
void main()
{
     printf("Helló világ\n");
}
```

A programot tetszőleges szövegszerkesztővel megírhatjuk. A forrásprogramot ezután le kell fordítani. A fordítás után keletkezett file már futtatható.

3 A C program szerkezete

Egy C program egy vagy több modulból áll. A modulok valamelyikének tartalmaznia kell egy `main` nevű függvényt. Ez lesz a program belépési pontja, vagyis ez kezd el végrehajtódni.
Modul:
Minden modul külön file. A modulok önállóan lefordíthatók, úgynevezett tárgykóddá. A tárgykódú modulokat az összeszerkesztő (linker) állítja össze egy programmá. (A linker általában egyben van a fordítóval, és kapcsolókkal szabályozható, hogy csak tárgykódot készítsen, vagy végezze el az összeszerkesztést is, vagy csak összeszerkesztést készítsen.)
Egy modulos program esetén is van dolga az összeszerkesztőnek, hiszen a modulunkban használt könyvtári függvények (a C nyelvhez adott előre megírt függvények, pl. `printf()`) egy vagy több tárgykódú modulban vannak elhelyezve.

Egy modul tartalmazhat:
- fordítónak szóló utasításokat (pl. `#include <stdio.h>`)
- deklarációkat, definíciókat (változó, típus, stb.)
- függvényeket (pl. `void main() { … }`)
- megjegyzéseket

3.1 Fordítás

A C nyelv tehát egy compileres nyelv.
A fordítás több menetben történik:
1. feldolgozza a fordítónak szóló utasításokat (előfordítás),
2. lefordítja a programot tárgykódúvá,
3. összeszerkeszti a többi modullal.

Eredménye egy futtatható file.

4 Általános szintaktikai elemek

4.1 Case-sensitive

A C nyelv case-sensitive, azaz érzékeny a kisbetű és nagybetű közötti különbségre.

4.2 Megjegyzések

Több soros megjegyzés:

```
/* megjegyzés szövege
    a következő sorban is folytatódik. */
```

Megjegyzés a sor végéig:

```
// megjegyzés
```

Egy megjegyzés lehet akár egy utasítás után is:

```
Utasítás;   //megjegyzés
```

4.3 Azonosítók

Azonosító állhat az angol ABC betűiből, számjegyekből, _-ból, de nem kezdődhet számjeggyel. A maximális hossza függ a C nyelv verziótól, de minimum 8 karakter. Nem lehet azonosító a nyelv egyik kulcsszava.

4.4 Operátorok

A C nyelvben nagyon sok operátor van. Ezeket nem kell mind megjegyezni. Először következzen az operátorok precedencia táblázata, majd külön bemutatom azokat, amelyeket ismerni kell. (Nem kell ismerni a bitléptető operátorokat, valamint a , operátort.)

<table>
<tr><td>1</td><td>(), [], ->, .</td><td></td></tr>
<tr><td>2</td><td>!, ~, ++, --, -, +, (típus), *, &</td><td>Jobbról balra</td></tr>
<tr><td>3</td><td>*, /, %</td><td></td></tr>
<tr><td>4</td><td>+, -</td><td></td></tr>
<tr><td>5</td><td>>>, <<</td><td></td></tr>
<tr><td>6</td><td><, <=, >, >=</td><td></td></tr>
<tr><td>7</td><td>==, !=</td><td></td></tr>
<tr><td>8</td><td>&</td><td></td></tr>
<tr><td>9</td><td>^</td><td></td></tr>
<tr><td>10</td><td>|</td><td></td></tr>
<tr><td>11</td><td>&&</td><td></td></tr>
<tr><td>12</td><td>||</td><td></td></tr>
<tr><td>13</td><td>?:</td><td>Jobbról balra</td></tr>
<tr><td>14</td><td>=, +=, -=, *=, /=, %=, |=, ^=, &=, >>=, <<=</td><td>Jobbról balra</td></tr>
<tr><td>15</td><td>,</td><td></td></tr>
</table>

Az 1. szint operátorai: elsődleges operátorok

`() operátor`: zárójelezés

`[] operátor`: tömbindexelés (lásd tömbök fejezet)

`-> operátor`: minősítés operátor, struktúrák és unionok mutatói esetén, (lásd struktúrák, unionok)

`.` `operátor`: minősítés operátor, struktúrák és unionok esetén (lásd struktúrák, unionok)

A 2. szint operátorai: egy operandusú operátorok

`!` `operátor`: logikai tagadás: `!kif` esetén, ha a `kif` értéke `0` (hamis), akkor az eredmény `1` (igaz), ha a `kif` értéke `nem 0` (igaz), akkor az eredmény `0` (hamis)

`~` `operátor`: bitenkénti tagadás

`++` `operátor`: növelő operátor: Létezik `++valt` (prefix) és `valt++` (postfix) is. A `valt++` kifejezés értéke `valt`, majd utána megnő a `valt` értéke. A `++valt` esetén `valt` értéke megnő és a kifejezés értéke `valt` új értéke lesz. Pl.

```
a=5;
b=a++;    //egyenértékű: b=a; a++; utasításokkal
```

utasítások után `a` értéke `6`, `b` értéke `5`,

```
a=5;
b=++a;    //egyenértékű: ++a; b=a; utasításokkal
```

utasítások után `a` értéke `6`, `b` értéke `6`.

A `++valt` kifejezést tartalmazó kifejezéseknek úgynevezett mellékhatásuk a `valt` megnövekedése. <u>Kerüljük a mellékhatásokat tartalmazó kifejezések használatát</u>, mert megnövekszik a hibázás lehetősége és csökken az olvashatóság. Pl.

```
b=++a+a++
```

Két mellékhatást is tartalmaz és nem is könnyű kideríteni mi az eredmény. (Nem beszélve az olyan olvashatatlan kifejezésről, mint pl `b=a++ + ++a;`)

<u>Biztonságos használni önálló utasításként (pl. `a++;` vagy `++a;`)</u>. Ilyenkor csak a mellékhatását használjuk ki, a kifejezés értékére nincs szükség, ezért mindegy, hogy a prefixet vagy a postfixet használjuk.

`--` `operátor`: csökkentő operátor: használata ugyanaz, mint a ++ operátoré, de növelés helyett csökkent 1-el.

`-` `operátor`: negatív előjel: `-kif` esetén a `kif` a `-1`-szeresére változik

`+` `operátor`: pozitív előjel: nincs hatása

`(típus)` `operátor`: típus konverziós operátor: `(típus)kif` esetén a `kif` típusát `típus`-ra alakítja át. Pl. `(int)(9.4/2.0)` kifejezés értéke `4` és típusa int

`*` `operátor`: indirekció (hivatkozás egy mutató által címzett területre, lásd mutatók)

`&` `operátor`: címképző operátor: `&valt` esetén az eredmény `valt` memóriacíme.

A 3. szint operátorai: multiplikatív operátorok

`*` `operátor`: szorzás: `kif1*kif2`

`/` `operátor`: osztás: `kif1/kif2`, ha mind a két kifejezés egész, akkor az eredmény is egész (lásd még típuskonverziók). Pl. `11/4` értéke `2` és `11/4.0` értéke `2.75`

`% operátor`: maradékképzés: kif1%kif2, mind a két kifejezésnek egésznek kell lennie. Pl. `11%4` értéke `3`

A 4. szint operátorai:

`+ operátor`: összeadás: `kif1+kif2`

`- operátor`: kivonás: `kif1-kif2`

Az 5. szint operátorai:

`<<, >> operátorok`: bitléptető operátorok

A 6. szint operátorai:

`<, <=, >, >= operátorok`: relációs operátorok (kisebb, kisebb egyenlő, nagyobb, nagyobb egyenlő): eredményük 0 hamis esetén, 1 igaz esetén. (Lehet használni így is, de nem célszerű: `b=5*(a<8);`)

A 7. szint operátorai:

`== operátor`: relációs operátor (egyenlőség): `kif1==kif2`, eredménye 0 hamis esetén, 1 igaz esetén. Nagyon veszélyes az értékadó operátorral (=) való összetévesztése. Pl. `a==5` eredménye, ha `a` értéke `5`, akkor `1 (igaz)`, különben `0 (hamis)`, `a=5` eredménye `5 (igaz)` ráadásul mellékhatásként `a` értéke megváltozik `5`-re.

`!= operátor`: relációs operátor (nem egyenlő): `kif1!=kif2`, eredménye 0 hamis esetén, 1 igaz esetén.

A 8. szint operátorai:

`& operátor`: bitenkénti és operátor

A 9. szint operátorai:

`^ operátor`: bitenkénti kizáró vagy operátor

A 10. szint operátorai:

`| operátor`: bitenkénti vagy operátor

A 11. szint operátorai:

`&& operátor`: logikai és operátor: `kif1 && kif2`, eredménye akkor igaz (nem 0), ha mind a két kifejezés igaz (nem 0), különben hamis (0). A kiértékelése rövidre zárt, azaz ha már az első kifejezés kiértékelése után el tudja dönteni az eredményt, akkor a második kifejezés már nem értékelődik ki. Ez csak akkor okozhat problémát, ha a második kifejezésben mellékhatás van (pl. ++ operátoros kifejezés).

A 12. szint operátorai:

`|| operátor`: logikai vagy operátor: `kif1 || kif2`, eredménye igaz (nem 0), ha bármelyik kifejezés igaz. Kiértékelése szintén rövidre zárt.

A 13. szint operátorai: (az egyetlen három operandusú operátor)

`?: operátor`: feltételes operátor: `kif1 ? kif2 : kif3`, eredménye `kif2`, ha `kif1` igaz (nem 0), és `kif3`, ha `kif1` hamis (0).

A 14. szint operátorai:

`= operátor`: értékadó operátor, `balérték = kif1`, ahol `balérték` lehet pl. változó, vagy egy mutató által mutatott terület. Az egész kifejezés eredménye `kif1` lesz, valamint mellékhatásként a `balérték` felveszi `kif1` értékét. A kifejezésnek csak a mellékhatását szokás használni, az értékét nem. Például

```
int a,b;
a = 5*6;    /*Mellékhatásként a változó felvette a 30 értéket, a kifejezés
            eredménye is 30, de ezt nem használjuk fel.*/
b = 2 * (a = 10);    /*Mellékhatásként a változó felveszi a 10 értéket,
            valamint (a=10) értéke is 10, a b értéke tehát 20 lesz és az
            egész kifejezés értéke is 20, de ezt már nem használjuk fel
            sehova. Ehelyett írjuk inkább két külön utasításba:
            a = 10;
            b = 2 * a;
```

`+=, -=, *=, …. operátorok`: értékadó operátorok. A `balérték op= kif1` egyenértékű a `balérték = balérték op kif1` művelettel. Például:

```
int a=5;
a += 2*5; //Az a értéke megnő 10-el. Vagyis 15 lesz.
```

A 15. szint operátorai:

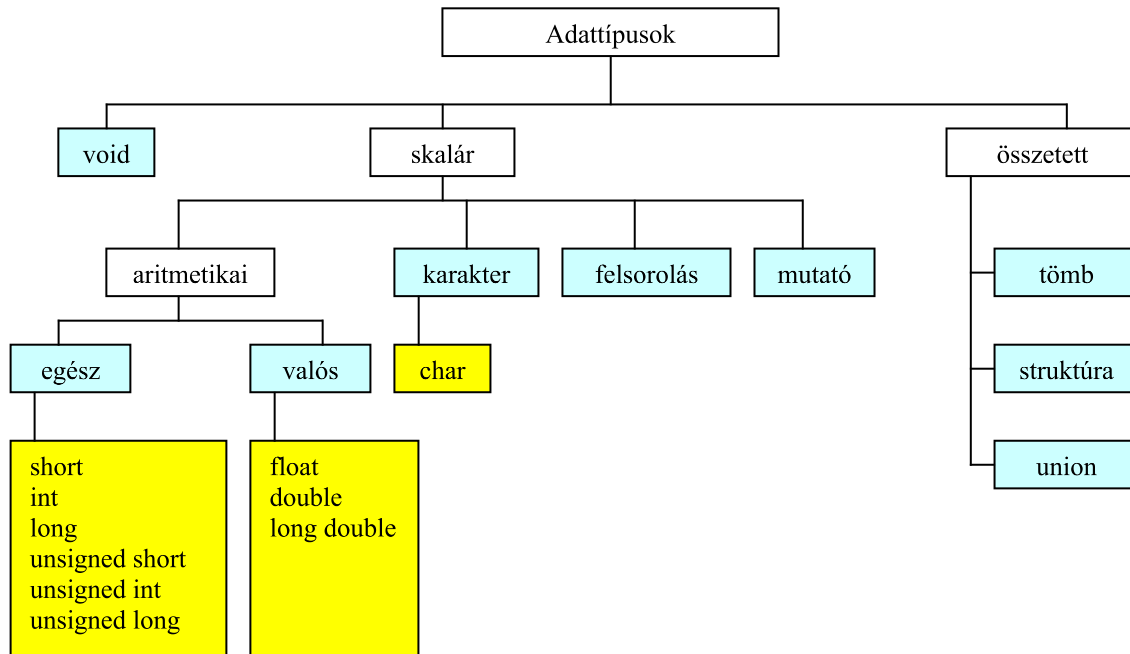
`, operátor`: vessző operátor

5 Adattípusok

A C nyelv adattípusok területén sok eltérést tartalmaz a többi nyelvhez képest.
Legfontosabb eltérések:

- A karakter típus teljesen egyenértékű az egésszel, csak megjelenítésükben különböznek. (Lehet például két karaktert osztani egymással.)
- Nincs külön logikai típus. Az egészeket használja logikai értékként. A 0 értéket hamisnak tekinti, az ettől eltérő egész értéket igaznak. A relációk eredménye pedig 0, ha hamis és 1 ha igaz.
- Nincs szöveges típus. A szövegek tárolására karakter tömböket használ. Szöveges konstans viszont létezik. Például ”ez egy szöveg”, ”tartalmazhat speciális \n karaktert is”, ”Ez egy \”idézőjeles\” szöveg”
- Létezik egy speciális tárolású struktúra típus az union.
- Nem létezik referencia típus.

A típusok összefoglaló táblázata:



Egészek:
1 byte-os előjeles: `short` (vagy `short int`)
1 byte-os előjeltelen: `unsigned short` (vagy `unsigned short int`)
2 byte-os előjeles: `int`
2 byte-os előjeltelen: `unsigned int`
4 byte-os előjeles: `long` (vagy `long int`)
4 byte-os előjeltelen: `unsigned long` (vagy `unsigned long int`)

Egész konstansok:
Decimális: `123, -35`
Oktális: `045` (tehát nullával kezdődik)
Hexadecimális: `0x4A` (tehát 0x-el kezdődik)
Előjeltelen tárolás előírása: `34u, 34U` (tehát a szám után egy `u` vagy `U` betű)

Valósak:
Kispontosságú: `float`
Duplapontosságú: `double`
Nagy pontosságú: `long double`

Valós konstansok:
Tizedespontos alak: `12.34, 0.4534`
Tudományos alak: `2.345e5, 0.56E-2` (azaz `234500.0, 0.0056`)
Alapértelmezés szerint duplapontosságú egy valós konstans, de elő lehet írni kispontosságot is a szám után tett `f` vagy `F` betűvel (`45.6F`).

Karakterek:
A C nyelvben a karakter típus tárolás és kezelés alapján valójában egy `unsigned short` típus, különbség csak a megjelenítésben van. Ebből következően egy karakter típusú adat

minden olyan helyen használható, ahol egyébként `unsigned short` használható lenne (azaz pl. lehet szorozni karaktereket).
Karakter: `char`
Karakter konstans: 'A', '-', '5'
Speciális karakter konstansok:

'\n': újsor
'\t': vízszintes tabulátor
'\f': lapdobás
'\'': aposztróf
'\0nn': az nn oktális kódú karakter
'\\': maga a \
'\r': kocsivissza
'\v': függőleges tabulátor
'\b': backspace
'\"': az "
'\xnn': az nn hexakódú karakter

Void:
A void egy speciális típus, ami két helyen használható:
- mutatók típusaként megadva jelzi, hogy még nem ismert a típus
- függvények visszatérési értékének típusaként megadva jelzi, hogy nincs visszatérési érték

A többi típust (felsorolás, mutató, tömb, struktúra, union) majd külön tárgyaljuk.

5.1 Típuskonverziók

A C nyelvben, mint minden nyelvben, szintén zajlanak típuskonverziók, ha egy műveletben különböző típusú adatok szerepelnek. Az általános szabály az, hogy a kisebb kapacitású konvertálódik a nagyobb kapacitásúra. C nyelvben bonyolítja a helyzetet az unsigned típusok jelenléte, hiszen ugyanannyi bitet használ mint a signed, mégsem konvertálható feltétlenül egyik sem a másikba. Bizonyos esetekben tehát mindkét operandus konvertálódik.

6 Deklarációk, definíciók

6.1 Változó deklaráció

A C nyelvben minden változót kötelező deklarálni!
Szintaktikája:

```
típus azonosítólista;
```

ahol az `azonosítólista` tetszőleges számú azonosító vesszővel elválasztva:

```
azonosító, azonosító, ….
```

Pl.

```
unsigned int jegy, darab, b;
double kerulet;
```

Változó deklaráció előfordulhat egy függvény belsejében, vagy a függvényeken kívül is. (Természetesen a változó hatáskörét és az élettartamát befolyásolja a deklaráció helye, lásd Hatáskörök, élettartamok.) A régebbi C verziók szerint a függvény belsejében történő deklarálás esetén a deklarációknak meg kell előzniük az utasításokat. Az újabb verziók, illetve a C++ esetén a függvény belsejében bárhol előfordulhat deklaráció.

Inicializálás
Lehetséges a változók inicializálása is.
Pl.

```
int b, osszeg=0;    //b nincs inicializálva, csak osszeg
```

6.2 Függvény definíció

Szintaktika:

```
típus név(paraméterek)
{
    utasítások
}
```

ahol a `típus` a visszatérési érték típusa, a `név` a függvény azonosítója, a paraméterek megadásának szintaktikája pedig:

```
típus paraméternév, típus paraméternév, …
```

Ha nincs paraméter a zárójeleket akkor is ki kell tenni!

```
típus név()
{
    utasítások
}
```

Ha nincs a függvénynek visszatérési értéke, azaz eljárás jellegű, akkor a `típus`nak void-nak kell lennie!

6.3 Függvény deklaráció

Bizonyos esetekben előfordulhat, hogy csak deklarálni kell egy függvényt. Pl. ha később definiálom, mint ahogy hivatkozok rá, vagy ha másik modulban van definiálva.
Szintaktikája:

```
típus név(paraméterek);
```

A programunkban használt könyvtári függvényeket is deklarálni kell. Szerencsére ezek deklarációját már előre elkészítették és egy-egy file-ba elhelyezték. Így ezeket csak be kell illesztenünk a saját programunkba.
A fordítónak szóló #include utasítással tetszőleges file-t beszúrhatunk a programunkba.

```
#include <filenév>
```

vagy

```
#include ”filenév”
```

Az első esetben útvonalat nem kell megadni, egy alapértelmezett helyen keresi a file-t. A másodikban útvonalat is meg kell adni.

6.4 Típus deklaráció

```
typedef típusleírás típusnév;
```

Például létrehozhatunk egy új típust, amely teljesen azonos az int típussal:

```
typedef int egesz;
```

ezek után:

```
egesz a;
```

Az struktúráknál, unionoknál és felsorolás típusnál vehetjük még hasznát. Lásd a megfelelő fejezeteket.

6.5 Elnevezett konstans deklarálása

A C nyelvben nincs külön lehetőség elnevezett konstansok létrehozására. Lehet viszont használni egy olyan fordítónak szóló utasítást, amely hasonló szerepkört tölt be.

```
#define VALAMI 45
```

Ennek hatására a fordító mindenhova, ahol a forrásprogramban ezután meglátja a `VALAMI` szót, behelyettesíti a `45`-öt.

7 Utasítások

7.1 Utasítások szintaktikája

- Utasítások csak függvény belsejében (függvény blokkjában) fordulhatnak elő!
- Az utasításokat `;` zárja.
- Az utasítás blokkok nyitó illetve záró jele a {}.

```
{
      utasítások
}
```

7.2 Utasítás fajták

7.2.1 Üres utasítás

```
;
```

Akkor használjuk, ha egy vezérlő szerkezet utasításrészébe nem akarunk írni utasítást.

7.2.2 Kifejezés utasítás

```
Kifejezés;
```

Leggyakoribb kifejezés utasítások az értékadó utasítás (lásd értékadó operátorok), növelő (lásd ++ operátor) vagy csökkentő (lásd -- operátor) utasítás, függvény hívás (lásd függvények).

Feladat:
Olvasson be egy egész számot, majd írja ki oktálisan, hexadecimálisan, karakteresen, és írja ki a négyzetgyökét 4 tizedes pontossággal.

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

void main()
{
     int szam;
     double gyok;

     printf("Kérek egy számot: ");
     scanf("%d", &szam);
     printf("A szám oktálisan: %o\n", szam);
     printf("A szám hexadecimálisan: %x\n", szam);
     printf("A %d kódú karakter: %c\n", szam, szam);
     gyok = sqrt(szam);
     printf("%d gyöke: %8.4lf\n", szam, gyok);
}
```

Az `stdio.h` a `printf, scanf` könyvtári függvények deklarációját tartalmazza. A `math.h` az `sqrt` könyvtári függvény deklarációját tartalmazza. A könyvtári függvények részletesebb ismertetése az utolsó fejezetben található.

7.2.3 Kétirányú elágazás

1. alak:

```
if (feltétel) utasítás;
```

A `feltétel` egy egész értékű kifejezés, ahol a 0 számít hamisnak, a nem 0 igaznak. A feltétel zárójelezése kötelező. Az `utasítás` csak egy darab utasítás lehet, vagy egy utasítás blokk. Az `utasítás` tartalmazhat akár vezérlő utasítást is (tehát lehet if utasításban újabb if vagy más vezérlő utasítás).
Szokásos írás:

```
if (feltétel)
    utasítás;
```

vagy utasításblokk esetén:

```
if (feltétel)
{
     utasítások
}
```

Szemantikája (jelentése, működése): Kiértékeli a feltételt, ha igaz (nem 0), akkor végrehajtja az utasítást, ha hamis (0), akkor nem.

2. alak:

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

A `feltétel` egy egész értékű kifejezés, ahol a 0 számít hamisnak, a nem 0 igaznak. A feltétel zárójelezése kötelező. Az `utasítás1` és `utasítás2` is csak egy darab utasítás lehet, vagy egy utasítás blokk. Az `utasítás` tartalmazhat akár vezérlő utasítást is (tehát lehet if utasításban újabb if).

Szokásos írás:
```
if (feltétel)
    utasítás1;
else
    utasítás2;
```
vagy utasítás blokkok esetén:
```
if (feltétel)
{
     utasítások
}
else
{
     utasítások
}
```

Szemantika: Kiértékelődik a `feltétel` és ha igaz (nem 0), akkor végrehajtódik `utasítás1`, különben pedig `utasítás2`.

Feladat:
Olvasson be három számot és írja ki a legnagyobbat.
```
#include <stdio.h>

void main()
{
    int a1, a2, a3;

    printf("Kérem az első számot: ");
    scanf("%d", &a1);
    printf("Kérem a második számot: ");
    scanf("%d", &a2);
    printf("Kérem a harmadik számot: ");
    scanf("%d", &a3);
    if (a1 > a2 && a1 > a3)
        printf("A legnagyobb: %d\n", a1);
    else
    {
        if (a2 > a3)
            printf("A legnagyobb: %d\n", a2);
        else
            printf("A legnagyobb: %d\n", a3);
    }
}
```
Egy másik megoldás:
```
#include <stdio.h>

void main()
{
    int a1, a2, a3;
    int max;

    printf("Kérem az első számot: ");
```

```
        scanf("%d", &a1);
        printf("Kérem a második számot: ");
        scanf("%d", &a2);
        printf("Kérem a harmadik számot: ");
        scanf("%d", &a3);
        max = a1;
        if (max < a2)
            max = a2;
        if (max < a3)
         max = a3;
        printf("A legnagyobb: %d\n", max);
    }
```

7.2.4 Többirányú elágazás

```
switch (kifejezés)
{
     case érték1:
          utasítás1;
     case érték2:
          utasítás2;
     .
     .
     default:
          utasításN;
}
```

A `kifejezés` egész értékű kell legyen. Az `érték1, érték2, …` konstans értékek, vagy konstans kifejezések (olyan kifejezés, amelynek minden operandusa konstans) lehetnek. Az `utasítás1, utasítás2, …, utasításN` egy utasítás vagy utasítás blokk lehet. A `default` rész elmaradhat.

Szemantika: Kiértékelődik a kifejezés, majd a végrehajtás attól az utasítástól folytatódik, amely előtt a kifejezés értékével megegyező érték áll. Ha egyikkel sem egyezik meg, akkor a default utáni utasításon folytatódik a végrehajtás. Ha egyikkel sem egyezik meg és nincs default rész, akkor egyik switch utasításba tartozó utasítás sem hajtódik végre.

Példa:

```
int a, b;
a=3;
switch (a)
{
     case 2:
          b=2;
     case 3:
          b=4;
     case 8:
          b=10;
}
```

A példában a kifejezés értéke a másodikként megadott értékkel egyezik tehát onnantól kezdve végrehajtja a switch-ben levő utasításokat. Azaz végrehajtódik a `b=4`, majd utána a `b=10` is.

Ha azt szeretnénk elérni, hogy csak az adott esethez tartozó utasítás hajtódjon végre, de a switch-ben utána következők nem, akkor használni kell a break utasítást.
Példa részlet:

```
int a, b;
a=3;
switch (a)
{
    case 2:
        b=2; break;
    case 3:
        b=4; break;
    case 8:
        b=10; break;
}
```

A példában a kifejezés értéke a másodikként megadott értékkel egyezik, tehát onnantól kezdve végrehajtja a switch-ben levő utasításokat. Azaz végrehajtódik a `b=4`, majd utána a `break` is. A `break` hatása az (lásd még break utasítás), hogy kilép a switch utasításból és a végrehajtás a switch-et követő utasításon folytatódik. A `b=10` tehát már nem hajtódik végre.

7.2.5 Elöltesztelő iteratív ciklus

```
while (feltétel) utasítás;
```

A `feltétel` egy egész értékű kifejezés, ahol a 0 számít hamisnak, a nem 0 igaznak. A feltétel zárójelezése kötelező. Az `utasítás` csak egy darab utasítás lehet, vagy egy utasítás blokk.
Szokásos írás:

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

vagy utasításblokk esetén:

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

Szemantikája: Kiértékeli a feltételt, ha igaz (nem 0), akkor végrehajtja az utasítást, majd visszaugrik a végrehajtás a feltétel kiértékelésre, ha hamis (0), akkor nem. Vagyis addig ismételgeti az utasítást, amíg igaz a feltétel. Mivel elöltesztelő ciklus, így ha már legelső feltétel kiértékeléskor hamis a feltétel, akkor az utasítás egyszer sem hajtódik végre.

Feladat:
Adja össze 1-től kezdve a pozitív egész számokat addig, amíg az összeg eléri a 10000-t. Hány számot adtunk össze?

```
#include <stdio.h>

void main()
{
    int kov = 0;
    int osszeg = 0;
```

```
        while (osszeg < 10000)
        {
            kov++;
            osszeg += kov;
        }
        osszeg-=kov;
        kov--;
        printf("%d számot adtunk össze.\n Az összeg:%d", kov);
    }
```
A ciklus addig számol, amíg az összeg már meghaladja a 10000-t, a ciklus után tehát az utolsó hozzáadott számot még levonjuk.

7.2.6 Hátultesztelő iteratív ciklus

```
    do utasítás while (feltétel);
```
A `feltétel` egy egész értékű kifejezés, ahol a 0 számít hamisnak, a nem 0 igaznak. A feltétel zárójelezése kötelező. Az `utasítás` csak egy darab utasítás lehet, vagy egy utasítás blokk.
Szokásos írás:
```
    do
         utasítás
    while (feltétel);
```
vagy utasításblokk esetén:
```
    do
    {
         utasítások
    } while (feltétel);
```

Szemantikája (jelentése, működése): Végrehajtja az utasítást, majd kiértékeli a feltételt. Ha igaz (nem 0), akkor ugrik vissza az utasítás végrehajtásra. Vagyis ez is addig ismételgeti az utasítást, amíg iga a feltétel, de mivel ez hátultesztelő ezért ha a feltétel már legelső alkalommal sem igaz, akkor is már egyszer végrehajtotta az utasítást.

Feladat:
Olvassunk be ellenőrzötten egy kisbetűt, azaz ha nem kisbetűt ad meg, akkor hibaüzenet ír ki és újra kéri. Írja ki a betű nagybetűs alakját.
```
    #include <stdio.h>

    void main()
    {
        char b;

        do {
            printf("Kérek egy kisbetűt: ");
            scanf("%c", &b);
            if (b<'a' || b>'z')
                printf("Ez nem kisbetű!\n");
        while (b<'a' || b>'z');
        printf("A nagybetűs alakja: %c\n", b-('a'-'A'));
```

```
}
```

7.2.7 Előltesztelő taxatívként használható iteratív ciklus

```
for (kif1; kif2; kif3) utasítás;
```

Végrehajtása teljesen megegyezik a következő utasítássorozattal:

```
kif1;
while (kif2)
{
     utasítás;
     kif3;
}
```

A `kif1` és `kif3` kifejezés utasítások kell legyenek. A `kif2` egy egész értékű kifejezés kell legyen. Ha `kif1`-et arra használjuk, hogy egy számláló változónak kezdő értéket adjunk, a `kif3`-t pedig arra használjuk, hogy ezt a változót növelgessük, akkor taxatívként használhatjuk a ciklust.

Feladat:
Írjuk ki egy beolvasott szám faktoriálisát!

```
#include <stdio.h>

void main()
{
    int szam;
    int j;
    long szorzat;
    printf(”A program egy megadott szám faktoriálisát
          számolja ki.\n”);
    printf(”Kérek egy számot: ”);
    scanf(”%d”, &szam);
    szorzat = 1;
    for (j=1; j<=szam; j++)
        szorzat *= j;
    printf(”%d!=%ld\n”, szam, szorzat);
}
```

7.2.8 Strukturált ugró utasítások

Break utasítás

```
break;
```

Csak ciklusok (while, do, for) és switch belsejében fordulhat elő. Hatására a ciklus vagy switch utáni utasításon folytatódik a végrehajtás. Több egymásba ágyazott ciklus (vagy switch) esetén csak a legbelsőből ugrik ki (abból, amelyik a break-t közvetlenül tartalmazza).

Continue utasítás

```
continue;
```

Csak ciklusok (while, do, for) belsejében fordulhat elő. Hatására a végrehajtás, azonnal a ciklus feltétel kiértékelésére ugrik.

Return utasítás
```
return;
```
vagy
```
return kifejezés;
```
Függvényekből való azonnali visszatérés. A `return` utáni kifejezés lesz a függvény visszatérési értéke. A `return` kifejezés nélkül, csak void visszatérési értékű, azaz eljárás jellegű függvényeknél használható.

7.2.9 Strukturálatlan ugró utasítás

Létezik ugyan a C nyelvben strukturálatlan ugró utasítás, de használatuk nagyon veszélyes, ezért óráinkon HASZNÁLATA TILOS!

8 Függvények

8.1 Jellegzetességek a C nyelvben

- a függvények és eljárások szintaktikája nem különbözik, azaz csak függvények léteznek, a void visszatérési érték típussal jelezzük, ha nem akarunk visszaadni értéket,
- nem lehet definiálni függvényt egy másik függvény belsejében
- bármelyik függvény meghívhat bármelyik függvényt, még saját magát is (rekurzív hívás), vagy akár a main függvényt is,
- a program belépési pontja a main függvény.
- visszatérési érték és paraméter nem lehet tömb, struktúra, union, lehet viszont ezekre mutató mutató

Szintaktika:
```
tipus fgvnev(paraméterek)
{
    fgv blokk
}
```

8.2 Paraméterátadás

Csak értékszerinti paraméterátadás létezik, tehát paraméterek csak akkor tudnak eredményt visszaadni, ha mutatót használunk (referencia típus nincs).
Szintaktikája:
```
típus név(típus param1, típus param2, …)
```
Nem lehetséges az azonos típusú paraméterek összevonása, mint más nyelvekben:
```
típus név(típus param1, param2)    //hibás
```
A paraméter nélküli függvények esetén is ki kell tenni a zárójeleket:
```
típus név()
```

8.3 A függvény meghívása

A void függvények csak önálló utasításként hívhatók meg. Az egyéb visszatérési értékű függvények meghívhatók önálló utasításként is, és kifejezés belsejében is.
Szintaktikája:

```
fgvnev(kif1, kif2, …)
```

ahol kif1, kif2, stb. kifejezések az aktuális paraméterek. Mivel az aktuális paraméterek értékei adódnak át rendre a formális paramétereknek (param1=kif1, param2=kif2, …), ezért ezek típusa olyan kell legyen, hogy az értékátadás megtörténhessen (lásd automatikus típuskonverziók).

8.4 Példák

1. Példa:
Készítsen egy függvényt, amely kap egy egész számot és visszaadja a szám faktoriálisát. Olvasson be egy egész számot és írja ki a faktoriálisát.

```
#include <stdio.h>

long fakt(int a)
{
    long szorzat;
    int j;

    for (j=1; j<=a; j++)
        szorzat *= j;
    return szorzat;
}

void main()
{
    int szam;
    long ered;

    printf(”A program egy megadott szám faktoriálisát
                számolja ki.\n”);
    printf(”Kérek egy számot: ”);
    scanf(”%d”, &szam);
    ered = fakt(szam);
    printf(”%d!=%ld\n”);
}
```

2. Példa:
Készítsen egy függvényt, amely három oldalhosszúságról megállapítja, alkothatnak-e háromszöget! Készítsen egy függvényt, amely a három oldalhosszúságból kiszámolja a háromszög területét! Készítsen egy függvényt, amely egy kapott valós számot 3tizedes pontossággal kiír az ”Az eredmény:” felirat után! Készítsen egy programot, amely beolvas három oldalhosszúságot és ha alkothatnak háromszöget akkor kiírja a területét, ha nem, akkor kiírja, hogy nem alkothat háromszöget.

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

```
int alkothate(double a, double b, double c)
{
    if (a+b<=c || a+c<=b || b+c<=a)
        return 0;
    return 1;
}

double teruletSzamit(double a, double b, double c)
{
    double s;
    s = (a+b+c)/2;
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

void kiir(double a)
{
    printf(”Az eredmény: %10.3lf\n”, a);
}

void main()
{
    double oldal1, oldal2, oldal3;
    printf(”Kérem az első oldalt: ”);
    scanf(”%lf”, &oldal1);
    printf(”Kérem a második oldalt: ”);
    scanf(”%lf”, &oldal2);
    printf(”Kérem a harmadik oldalt: ”);
    scanf(”%lf”, &oldal3);
    if (alkothate(oldal1, oldal2, oldal3))
    {
        ter=teruletSzamit(oldal1, oldal2, oldal3);
        kiir(ter);
    }
    else
        printf(”Nem alkot háromszöget!\n”);
}
```

9 Hatáskörök, élettartamok

9.1 Függvények hatásköre

A függvényeket a fordítónak ismernie kell akkorra, mire a meghívásuk megtörténik. Vagyis a függvények csak a deklarálás helyétől kezdve ismertek. Ezért nem mindegy, hogy a függvényeket milyen sorrendben definiáljuk. Például

```
void fgv()
{ masik(); }
```

```
void masik()
{ … }
```

Hibaüzenetet kapunk, mert a `masik` függvény hívásakor még nem ismert a fordító számára a függvény. Ha a két függvény írási sorrendjét megcseréljük, akkor már rendben van. Vannak esetek, amikor semmiképpen sem tudjuk olyan sorrendben definiálni a függvényeket, hogy egy függvény mindig csak előtte definiált függvényt hívjon, ilyenkor a később definiált függvényt deklarálnunk kell még a hívás előtt. Például:

```
void masik();

void fgv()
{ masik(); }

void masik()
{ … }
```

Egy függvény lehet

- modulra lokális. A modulra lokális csak abban a modulban elérhető, amelyben definiálták. Más modulokból nem elérhető. A függvény definiálásakor a `static` kulcsszót kell tenni a függvény fejléce elé.
  ```
  static void fgv()
  { … }
  ```
- globális. A globális függvény bármelyik modulból elérhető. Mivel továbbra is igaz, hogy a függvény használatához a fordítónak ismernie kell a függvény prototípusát (a fejlécben megtalálható információkat: név, paraméterek, visszatérési érték típus), ezért minden olyan modulban, ahol használni akarom a másik modulban definiált függvényt, deklarálnom kell. Az ilyen deklarációk elé ki kell tenni az `extern` kulcsszót, jelezve azt, hogy a függvény nem ebben a modulban lesz definiálva. Például
  Egyik modul:
  ```
  void fgv()     //globális függvény, nincs előtte static
  { … }
  ```
  Másik modul:
  ```
  extern void fgv();

  void hivo()
  { fgv(); }
  ```

9.2 *Változók hatásköre és élettartama*

A változók definiálásuk helyétől függően két csoportra bonthatók:

- függvényeken kívül deklarált változók. A függvényeken kívül deklarált változók szintén lehetnek globálisak, illetve modulra lokálisak. Ezekre ugyanazok mondhatók el, mint a függvények esetén. A program olvashatóságát erősen rontják, tehát használatuk kerülendő!

- függvényeken belül deklarált változók. Hatáskörük a definiálás helyétől annak a blokknak a végéig tart, amelyben definiálták. Hatáskörük kiterjed a belső blokkokra is.

```
{
        …           //a nem ismert (C nyelvben itt csak deklarációk lehetnek,
                    hiszen az összes deklaráció a függvény blokk elején kell legyen)
        int a;
        …           //a ismert
        {
              …     //a ismert
        }
        …           //a ismert
}
```

Élettartam szerint két fajtája van:

  - dinamikus: a definiáláskor jön létre és a blokk elhagyásával el is pusztul. A blokkba visszatéréskor újból létrejön.

```
{
      auto int b;    //az auto az alapértelmezés
      int a=0;       //tehát így is ugyanaz
      …
}
```

  - Statikus: a definiáláskor jön létre, de a blokk elhagyásával nem pusztul el. Bár a blokkon kívül nem elérhető, de a blokkba visszatérve újból használható, előző értéke megmarad.

```
{
      static int c=0;     //statikus, csak első
                          //alkalommal jön létre
      int a=0;            //dinamikus, minden
                          blokkba //lépéskor létrejön
      a++;       //a értéke 1 lesz
      c++;       //c értéke annyi lesz, ahányszor belép
      …          //a blokkba
}
```

9.3 Paraméterek hatásköre és élettartama

A függvények paraméterei mindig dinamikusak. Függvény meghívásakor létrejönnek, és függvény elhagyásakor elpusztulnak. Hatáskörük a függvény blokkjára lokális.

9.4 Azonosító elfedése

Megengedett, hogy egy blokkban olyan azonosítóval definiáljunk változót, amilyen már van egy külső blokkban (a külső blokkban definiált változó érvényes ebben a blokkban is). Ilyenkor mindig a szűkebb hatáskörű változó érhető csak el.

10 Tömb

A C nyelvben csak statikus tömbök vannak. Azaz a definícióban meg kell adni az elemek számát.

```
típus név[elemszám]
```

A `típus` lehet akár összetett típus is. Az `elemszám` csak konstans kifejezés lehet. Pl.

```
int a[5];
```

Hivatkozás az elemekre:

```
név[sorszám]
```

Az elemek sorszámozása 0-val kezdődik, és (elemszám-1)-ig tart. A sorszám megadása tetszőleges egész típusú kifejezéssel megtörténhet. Ha a sorszám kifejezés értéke nagyobb, mint az elemszám-1, akkor nem kapunk hibaüzenetet, a program hibásan fog működni.

Példák:

1. Példa:

Olvasson be 10 darab egész számot és írja ki az átlagukat, valamint a legnagyobb számot.

```
#include <stdio.h>

void main()
{
    int j;
    int szamok[10];
    int osszeg;
    int maxi;

    for (j=0; j<10; j++)
    {
        printf(”Kérem a %d. számot: ”, j+1);
        scanf(”%d”, &szamok[j]);
    }
    osszeg = 0;
    for (j=0; j<10; j++)
        osszeg += szamok[j];
    printf(”A számok átlaga: %lf\n”, osszeg/10.0);
    maxi=1;
    for (j=0; j<10; j++)
        if (szamok[maxi]<szamok[j])
            maxi=j;
    printf(”A legnagyobb szám: %d\n”, szamok[maxi]);
}
```

2. Példa:

Készítsünk egy függvényt, amely megkap egy nettó értéket és egy áfakulcsot, és visszaadja a bruttó értéket. Olvassunk be nettó árakat és vásárolt darabszámokat, amíg nullát nem írunk be árnak (maximum 20 árucikk lehet), határozzuk meg a bruttó fizetendő összeget. Csak egész forintokkal dolgozzunk, az egyszerűség kedvéért nem kerekítünk, hanem csonkolunk. Az áfakulcs minden terméknél 25%.

```
#include <stdio.h>

long bruttosit(long netto, int afakulcs)
{
    return (int)(netto+netto*afakulcs/100.0);
```

```
}

void main()
{
    long nettoArak[20];
    int darabok[20];
    long beolvasott;
    long brutto;
    long bruttoOsszeg;
    int arukSzama;

    arukSzama = 0;
    do
    {
        printf(”Kérem a %d. áru nettó egységárát: ”,
                    arukSzama+1);
        scanf(”%ld”, &beolvasott);
        if (beolvasott != 0)
        {
            nettoArak[arukSzama] = beolvasott;
            printf(”Kérem a %d. áru darabszámát: ”,
                    arukSzama+1);
            scanf(”%d”, &darakok[arukSzama]);
            arukSzama++;
        }
    } while (beolvasott != 0);

    for (j=0; j<arukSzama; j++)
    {
        brutto = bruttosit(nettoArak[j]*darabok[j], 25);
        bruttoOsszeg += brutto;
    }
    printf(”A végösszeg: %ld\n”, bruttoOsszeg);
}
```

11 Mutató

A mutató olyan változó, amely értéke egy memóriacím. Léteznek operátorok, amelyekkel a mutató által mutatott memóriaterületen levő adatot lehet elérni. A mutatónak ugyanolyan fontos adata a típusa, mint egy közönséges változónak. A mutató típusa a mutatott terület típusát jelenti.

11.1 Mutató deklarációja:

```
típus *változónév;
```

11.2 Műveletek mutatókkal

11.2.1 Értékadás mutatónak:

1. Egy meglévő területünk címét adjuk értékül a mutatónak

`int a;`	//definiálunk egy változót (lefoglalódik neki egy memóriaterület, értéke még nincs)
`int *m;`	//definiálunk egy mutatót (lefoglalódik neki egy memóriaterület, értéke még nincs, azaz nem mutat sehova)
`m=&a;`	//értéket adunk a mutatónak, értéke az „`a`" címe lesz (& címképző operátor)

2. Dinamikusan foglalunk egy területet és a lefoglalt terület kezdőcímét adjuk értékül a mutatónak. A dinamikusan foglalt területet a mutató segítségével fogjuk tudni elérni. Ez nem tananyaga ennek a félévnek.

11.2.2 Hivatkozás a mutatott területre

A mutatott területre a mutató segítségével is tudunk hivatkozni. Erre szolgál az indirekciós operátor. Például:

```
int a = 10;
int *m;
m=&a;
printf(”Az a változó értéke: %d”, *m);
*m=*m+1;
printf(”Az a változó értéke: %d”, a);
```

Nagyon veszélyes hibát eredményezhet, ha olyan mutató esetén használjuk az indirekciós operátort, amely még nem kapott értéket, azaz nem mutat sehova. Hiszen a mutatóra is igaz, hogy ha még nem kapott értéket, akkor az értéke véletlenszerű (pontosabban nem tudhatjuk mi az értéke). A lenti két utasítással egy véletlenszerű memóriaterület tartalmát írtuk át.

```
int *m;
*m=34;
```

11.2.3 Összeadás, kivonás

A mutatókhoz hozzá lehet adni egy egész számot, vagy ki lehet vonni egy egész számot belőle. Például, ha p egy double típusú mutató (double *p;) és p értéke 12000 (a 12000-es memóriacímre mutat), akkor p+2 hatására az értéke 2 double-nyival odébb fog mutatni (azaz a 12016-ra, hiszen egy double 8 byte). Ez nagyon jól kihasználható tömbök esetén, hiszen ott az adatok egymás után helyezkednek el a memóriában.

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

`b=&a[0];`	b most az a[0]-ra mutat
`c=*b;`	c változó a[0] értékét veszi fel
`c=*(b+4);`	c változó a[4] értékét veszi fel
`b=b+2;`	b most az a[2]-re mutat
`c=*b;`	c változó a[2] értékét veszi fel

11.2.4 Indexelés

A mutatókra is használható az index operátor. Például egy double típusú mutató esetén (double *p;), p[0] megegyezik a *(p+0)-val, p[1] megegyezik *(p+1)-el, stb.

```
int a[10];
int *b;
b=&a[0];
```

`b[0]=5;` azonos a *p=5 utasítással, vagyis a[0] értéke lesz 5

`b[1]=8;` azonos a *(p+1)=7 utasítással, vagyis az a[1] értéke lesz 8

`b[6]=9;` azonos a *(p+6)=9 utasítással, vagyis a[6] értéke lesz 9

11.3 Mutató, mint függvényparaméter

Mivel nincs címszerinti paraméterátadás, így csak mutatókkal lehet megoldani, hogy egy függvény megváltoztassa a hívónak egy változóját.

Példa:

Készítsen egy függvényt, amely megcseréli kér változó értékét. Készítsen egy programot, amelyben beolvas 10 egész számot egy tömbbe, majd megkeresi a legkisebbet, és legelőre rakja.

```
#include <stdio.h>

void csere(int *a, int *b)
{
    int seged;
    seged = *a;
    *a = *b;
    *b = seged;
}

void main()
{
    int szamok[10];
    int j;
    int mini;

    for (j=0; j<10; j++)
    {
        printf(”Kérem a %d. számot: ”, j);
        scanf(”%d”, &szamok[j]);
    }
    mini = 0;
    for (j=0; j<10; j++)
        if (szamok[mini]>szamok[j])
            mini=j;
    csere(&szamok[mini], &szamok[1]);
    for (j=0; j<10; j++)
        printf(”%d ”, szamok[j]);
}
```

11.4 Mutató, mint visszatérési érték

Függvény visszatérési értéke is lehet mutató.
típus *fgvneve(paraméterek)
Figyelni kell arra, hogy ne adjuk vissza a címét egy auto-ként definiált lokális változónak, hiszen ennek a területe felszabadul a függvényt elhagyva.

```
int *kisebbCimeHibas(int a, int b)
{
    int c;
    if (a<b)
        c=a;
    else
        c=b;
    return &c;
}
```

A fenti függvény hibás, hiszen egy olyan változónak adja vissza a címét, ami megszűnik a függvényt elhagyva. Ugyanígy hibás lenne valamelyik paraméter címét visszaadni, hiszen azok is auto típusú változók.

```
int *kisebbCimeJo(int *a, int *b)
{
    if (*a<*b)
        return a;
    return b;
}
```

Ennél a megoldásnál valamelyik paraméter értékét adom vissza, ami viszont a hívó függvény egy változójának a címét tartalmazza.

11.5 Tömb címe

A tömb kezdetének címét megkaphatjuk úgyis, hogy vesszük a 0.-ik elem címét (&a[0]), de a tömb neve is tartalmazza a tömb kezdőcímét. Tehát a fenti példa első három sora így is kinézhetett volna:

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

11.6 Tömb paraméter, és visszatérési érték

Tömböt nem lehet átadni függvénynek, át lehet viszont adni a tömb kezdőcímét egy mutatónak. Mivel a mutató esetén is lehet indexelést használni, ezért a függvényen belül a mutatót úgy kezelhetem, mintha tömb volna.
Példa:
Készítsen egy függvényt, amely egy valós tömb legnagyobb elemének az indexét adja vissza. Készítsen egy programot, amely beolvas 10 db valós számot és kiírja a legnagyobbat.

```
#include <stdio.h>

int maxindex(double *tomb, int elemszam)
{
    int j;
    int maxi = 1;
    for (j=0; j<elemszam; j++)
```

```
            if (tomb[maxi]<tomb[j])
                maxi=j;
        return maxi;
    }

    void main()
    {
        double szamok[10];
        int j;
        int maxi;

        for (j=0; j<10; j++)
        {
            printf("Kérem a %d. számot: ", j);
            scanf("%lf",szamok[j]);
        }
        maxi = maxindex(szamok, 10);
        printf("A legnagyobb szám: %lf\n", szamok[maxi]);
    }
```

12 Szöveges adatok

Mivel szöveges adattípus nincs, ezért a szövegeket karakter tömbökben tárolhatjuk. Hogy a tömb meddig van feltöltve azt egy végére berakott nulla kódú karakter jelzi. Azok a függvények, amelyek karaktertömbökön végeznek műveleteket, a záró karakterből tudják meddig tart a szöveg.

```
char valami[10];
valami[0]='a';
valami[1]='l';
valami[2]='m';
valami[3]='a';
valami[4]='\0';
printf("A szöveg: %s\n", valami);
```

Ha a záró karaktert nem adjuk meg a printf függvény egészen addig írja ki a memória tartalmát, amíg egy záró karaktert nem talál.

Mivel kényelmetlen így értéket adni egy karaktertömbnek, ezért léteznek előre definiált könyvtári függvények erre. (Includálni kell a string.h-t.)

```
char valami[10];
strcpy(valami, "alma");
```

Szövegek beolvasására is használható a scanf függvény. Csak egy szót lehet beolvasni vele. A beolvasott szöveg végére automatikusan elhelyezi a záró karaktert.

```
char s[40];
printf("Kérek egy szót: ");
scanf("%s",s);
printf("A beírt szó: %s\n", s);
```

Példa:

Készítsünk egy függvényt, amely egy paraméterként kapott szöveg hosszát adja vissza! (Ilyen egyébként létezik már strlen a neve.)
Készítsünk egy függvényt, amely kap egy szöveget és egy betűt, majd –1-et ad, ha a szövegben nem található meg a betű és az első előfordulás helyét adja, ha megtalálható.
Készítsen egy programot, amely beolvas egy szót, majd a szóban előforduló első 'a' betűt lecseréli 'b'-re.

```
#include <stdio.h>

int hossz(char *szo)
{
    int j;
    for (j=0; szo[j]!='\0'; j++);
    return j;
}

int holvan(char *szo, char betu)
{
    int j;
    int h;
    int hol;

    h=hossz(szo);
    hol = -1;
    for(j=0; j<h; j++)
    {
        if (szo[j]==betu)
        {
            hol=j;
            break;
        }
    }
    return hol;
}

void main()
{
    char s[50];
    int hol;

    printf("Kérek egy szót: ");
    scanf("%s", s);
    hol = holvan(s, 'a');
    if (hol != -1)
        s[hol] = 'b';
    printf("A változtatott szöveg: %s\n", s);
}
```

13 Felsorolás

A felsorolás típus egy programozható által definiálható típus. Előbb definiálni kell az új típust, majd utána felhasználhatjuk az új típusunkat a változó deklarációkban.
Típus definíció:

```
enum típusnév {elem1, elem2, …};
```

Változó deklaráció:

```
enum típusnév változónév;
```

Kényelmesebb megoldást eredményez a `typedef` használata.

```
typedef enum {elem1, elem2, …} típusnév;
típusnév változónév;
```

A C nyelv a felsorolás típust egészeknek tekinti. Az elemekhez a felsorolás sorrendjében egy egészet rendel hozzá. A felsorolás típusú adatokkal ezért minden olyan művelet elvégezhető, amely egészekkel elvégezhető.
Pl.

```
enum MunkaNapok {Hetfo, Kedd, Szerda, Csutortok, Pentek};
enum MunkaNapok egynap;
enum MunkaNapok masiknap;
egynap = Kedd       //érvényes
egynap = 2          //ua.
masiknap = egynap;  //érvényes
egynap = 9          //érvénytelen
```

14 Struktúra

Típus definíció:

```
struct típusnév {
    Mező deklarációk
};
```

Változó deklaráció:

```
struct típusnév változónév;
```

Például:

```
struct AruTipus {
    char nev[50];
    int ar;
};
struct AruTipus aru1;
```

vagy a `typedef` használatával:

```
typedef struct {
    char nev[50];
    int ar;
} AruTipus2;
AruTipus2 aru2;
```

Hivatkozás a struktúrák tagjaira:

```
változó.mező
```

vagy

```
struktúramutató->mező
```

Például:

```
AruTipus2 a, *b;
a.ar = 100;
b = &a;
b -> ar = 200;
```

Példa:
Készítsen egy függvényt, amely egy paraméterként kapott Termek típusú struktúra értékeit írja ki. Készítsen egy programot, amely beolvas két termék nevét és árát, majd kiírja a drágább termék adatait.

```
#include <stdio.h>
typedef struct {
    char nev[20];
    int ar;
} Termek;

void kiir(Termek *a)
{
    printf(”%s ára %d\n”, a->nev, a->ar);
}

void main()
{
    Termek t1, t2;

    printf(”Kérem az első termék nevét: ”);
    scanf(”%s”, t1.nev);
    printf(”Kérem az első termék árát: ”);
    scanf(”%d”, &t1.ar);
    printf(”Kérem a második termék nevét: ”);
    scanf(”%s”, t2.nev);
    printf(”Kérem a második termék árát: ”);
    scanf(”%d”, &t2.ar);
    if (t1.ar < t2.ar)
        kiir(&t1);
    else
        kiir(&t2);
}
```

15 Union

Az union deklarációjának szintaktikája szinte teljesen megegyezik a struktúráéval, csak `struct` helyett az `union` kulcsszót kell írni. Az union tagjaira való hivatkozás is teljesen megegyezik a struktúráéval. A különbség a tárolásukban van. A struktúra tagjai a memóriában egymást követően helyezkednek el. Az union tagjai viszont egymáson. Az union minden tagja ugyanazon a memóriacímen kezdődik, legfeljebb nem egyforma hosszúak. Vagyis az union

bármelyik tagjának megváltoztatom az értékét, azzal az union összes többi tagjának megváltoztattam az értékét.

16 Függvény könyvtárak

A C nyelvű programozásban nagy segítséget jelentenek az előre elkészített függvények. A leggyakrabban használtakat nézzük meg ebben a fejezetben.

16.1 Kiírás, beolvasás

A C nyelvben könyvtári függvények léteznek a kiírásra, beolvasásra.
A két leggyakoribb függvény a `printf, scanf` használatát célszerű megjegyezni.

A printf
Általános alak:

```
printf(formátumstring, értéklista);
```

A `formátumstring` tartalma lehet: normál karakterek, formátumvezérlések
A formátumvezérlések adják meg a kiírandó adatok kiírási formáját.
A formátum vezérlések legegyszerűbb alakja:

```
%x
```

x helyén a kiírandó érték típusától függően a következők állhatnak:

c: karakter
d: előjeles egész (int), decimálisan konvertálva
ld: hosszú előjeles egész (long), decimálisan
u: előjel nélküli egész (unsigned int), decimálisan
o: előjel nélküli egész (unsigned int), oktálisan
x: előjel nélküli egész (unsigned int), hexadecimálisan
f: lebegőpontos szám (float), tizedespontos alakban
lf: duplapontos lebegőpontos szám (double)
Lf: nagy pontosságú lebegőpontos szám (long double)
e: lebegőpontos szám (float), exponenciális alakban
s: string (char[])

Pl.

```
int a=5;
double b=0.5;
printf(”Az egyik érték %d volt, míg a másik %lf volt.\n”, a, b);
```

esetén a kiírás:

```
Az egyik érték 5 volt, míg a másik 0.5E0 volt.
```

<u>Nagyon fontos, hogy annyi formátumvezérlés legyen, ahány kiírandó adat, valamint a kiírandó adatnak megfelelő formátumvezérlést használjunk!!!</u>

A % jel és a típust jelző betűk közé lehet még tenni néhány jelet.
Pl.

`%15d` : 15 karakter szélességben jobbra igazítva írja ki az adatot
`%-15d` : 15 karakter szélességben balra igazítva írja ki az adatot
`%+d` : mindig kiírja az előjelet, akkor is, ha pozitív

`%15.4f` : 15 karakter szélességben, és 4 tizedes jegy pontossággal írja ki az adatot

A scanf

Általános alak:

```
scanf(formátumstring, változócímek);
```

A formátumstring most kizárólag formátumvezérléseket tartalmazhat. A formátumvezérlések alakja hasonló a printf-éhez.

```
%x
```

ahol az x:

d: int
ld: long
hd: short
e, f, g: float
lf: double
Lf: long double
o: int oktálisan
x: hexadecimális int
u: unsigned int
c: karakter
s: string
[...]: csak a megadott karakterek mint input

A % és a betűjel közé itt csak egy számot lehet tenni, ami a maximum beolvasott karakterek számát jelenti.

Nagyon fontos, hogy a változóknak (amibe eltárolja a beolvasott értékeket) a címét kell megadni. Nem mutató típusú változó esetén használni kell a címképző operátort a változó címének előállításához (&).

Pl.

```
int a;
printf(”Kérem a változó értékét:”);
scanf(”%d”, &a);
```

16.2 Matematikai függvények

A math.h header file-ban vannak deklarálva. Csak felsorolásszerűen (a nevek legtöbbször elmondják mire való):
sin, cos, tan, asin, acos, atan, abs, exp, log, log10, pow (hatványozás), sqrt (gyökvonás), ceil (felkerekítés), floor (lekerekítés), sinh, cosh, tanh (hiperbólikuszok), fmod (osztás maradék), modf (egész részre és törtrészre bontás), stb. Konstansok is vannak például M_PI, M_E.

16.3 Egyéb hasznos könyvtári függvények

A string.h header file-ban vannak deklarálva a szöveges adatokon manipuláló függvények. Például strcpy (szöveg bemásolása egy karaktertömbbe), strlen (szöveg hossza), strchr (karakter keresése szövegben), strcat (szöveg összefűzés), strcmp (szövegek összehasonlítása), stb.
A stdlib.h header file-ban van például: rand (véletlenszám generálás), atof, atoi (szövegből konvertálás számmá).