1 Tartalomjegyzék

1 TARTALOMJEGYZÉK 1

2 C PROGRAMOZÁS KÖZÉPISKOLÁSOKNAK 2

2.1 AZ ELSŐ C PROGRAM 2
2.2 MEGJEGYZÉSEK 2
2.3 A VÁLTOZÓKRÓL 2
2.3.1 Felhasználói típus 4
2.4 KONSTANSOK 5
2.4.1 A const kulcsszó hazsnálatával 5
2.4.2 Az előfordítónak szóló helyettesítéssel 5
2.5 OPERÁTOROK 5
2.5.1 Aritmetikai operátorok 6
2.5.2 Összehasonlító és logikai operátorok 6
2.5.3 Léptető operátorok 6
2.5.4 Bitműveletek 6
2.5.5 Értékadó operátorok 7
2.5.6 Feltételes operátor 7
2.5.7 Pointer operátorok 7
2.5.8 Típuskonverziók 7
2.6 ADATOK BEOLVASÁSA A BILLENTYŰZETRŐL 8
2.7 ITERÁCIÓK 9
2.7.1 while ciklus 9
2.7.2 for ciklus 10
2.7.3 do-while ciklus 11
2.8 SZELEKCIÓK 11
2.8.1 if utasítás 11
2.8.2 if-else szerkezet 12
2.8.3 switch utasítás 12
2.9 TÖMBÖK 13
2.9.1 Egydimenziós tömbök 13
2.9.2 Sztringek 14
2.9.3 Kettő és több dimenziós tömbök 16
2.10 FELHASZNÁLÓ ÁLTAL DEFINIÁLT TÍPUSOK 17
2.10.1 Struktúrák 17
2.10.2 Struktúrát tartalmazó tömbök 17
2.11 FÜGGVÉNYEK 18
2.11.1 Paraméterátadás 18
2.11.2 Lokális és globális változók 19
2.11.3 Automatikus és statikus változók 19
2.11.4 Register módosító jelző 20
2.11.5 Vektor átadása függvénynek 20
2.12 MUTATÓK (POINTEREK) 21
2.12.1 Vektorok és mutatók 21
2.12.2 Kétdimenziós tömbök és pointerek 23
2.12.2.1 Dinamikus tömb egy vektorban 23
2.12.2.2 Konstansban megadott tömbméret, dinamikus helyfoglalással 24
2.12.2.3 Mutatótömb 25
2.12.3 Struktúrák és mutatók 25
2.12.4 Dinamikus lista 26
2.13 MAKRÓK 28
2.13.1 Függvényszerű makrók 28

2 C programozás középiskolásoknak

2.1 Az első C program

Tekintsük a következő egyszerű C nyelven írt programot:

```
#include <stdio.h>

main()
  {
    printf(" Első C programom \n");
  }
```

A program kimenete az idézőjelek között szereplő felirat, a kurzor pedig a következő soron áll.

Nézzünk néhány megjegyzést a fenti programmal kapcsolatban:

1. A C programozási nyelv különbséget tesz a kis és a nagy betűk között. Minden C parancsnak kis betűsnek kell lennie
2. A C program belépési pontját a
   `main()`
   függvényhívás azonosítja. Egyelőre a függvényt argumentumok nélkül hívtuk meg, a későbbiekben ennek részletezésére még kitérünk.
3. A { és } a kezdő és a végpontját jelölik a végrehajtási résznek.
4. `#include <stdio.h>` nélkül nem működne a `printf()` függvény.
5. A `printf()`-ben a kiírandó szöveget dupla idézőjelek közé kell tennünk. Vegyük észre azt is, hogy a \n karakter nem került kiírásra. Tehát a `printf()` külön tartalmazza a kiírandó szöveget és a kiírást befolyásoló változókat. Ami a dupla idézőjelek között megjelenik változtatás nélkül kiírásra kerül, kivétel ez alól a \ és % karaktereket követő jel, vagy a jelsorozat. A \n változó a fordító számára azt jelenti, hogy a következő karakter kiírása előtt új sort kell kezdenie
6. A parancsokat a ; zárja le.

2.2 Megjegyzések

A megjegyzéseket a programszöveg mayarázatainak beírására szoktuk használni. Hibakereséskor is jó hasznát vehetjük azonban, ha a program egy részének végrehajtását szeretnénk kihagyni, akkor azt megjegyzés blokkba zárhatjuk.

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

2.3 A változókról

A C típusos programnyelv. Ez azt jelenti, hogy mielőtt egy változót használni szeretnénk deklarálnunk kell azt. Figyeljünk arra, hogy a a fordítóprogram a változó nevekben is különbséget tesz a kis és a nagy betűk között.

A névre a következő megkötések érvényesek:

- csak betűket, számjegyeket és aláhúzás karaktert tartalmazhat
- betűvel kell kezdődnie
- hossza legfeljebb 32 karakter (implementáció függő)

A változó deklaráció szabálya C-ben a következő:

```
Típus nev1,nev2,…;
```

Például:

```
int s;
float f,k;
```

Egy változót kezdőértékkel is elláthatunk.

```
int s=10;
```

Nézzük a következő C programot:

```
#include <stdio.h>

main()
{
   int sum;

   sum = 500 + 15;
   printf("500 és 15 összege %d\n", sum);
}
```

A programban létrehoztunk egy egész típusú változót, majd egy összeget rendeltünk hozzá. Figyeljük meg a változó kiíratását! Az idézőjelek között most találkozunk először a % jellel. Mögötte kell megadnunk, hogy erre a helyre milyen típusú változó kerül kiírásra. A d betű az egész típusra utal. Az idézőjel után következik a változó neve. Ennek értéke kerül majd beírásra a szövegbe.

Az adatoknak a C-ben négy alaptípusa van: egész (int), karakter (char), valós (float, double),

Az alaptípusokat elláthatjuk módosító jelzőkkel is, ekkor az értékkészlet módosul.

Pl.

```
int a
unsigned[1] int b
```

Az első esetben az a értéke –32768 és 32767 között lehet, míg a második esetben a b értéke 0 és 65535 közötti szám lehet. A signed[2] módosító jelző is használható, de alapértelmezés szerint minden egész változó ilyen lesz.

[1] Előjel nélküli
[2] Előjeles

A következő táblázatban felsoroltunk néhány típust, értékkészletükkel együtt.

Adattípus	Értékkészlet	Méret (byte)	Pontosság(jegy)
char	-128..127	1	
unsigned char	0..255	1	
int	-32768..32767	2	
unsigned int	0..65535	2	
long int	-2147483648..2147483647	4	
unsigned long int	0..4294967295	4	
float	3.4e-38..3.8e+38	4	6
double	1.7e-308..1.7e+38	8	15
long double	3.4e-4932..3.4e+4932	10	19

A printf() függvényben a változókat csak úgy tudjuk kiíratni, hogy az idézőjelek között % jel után megadjuk a változó típusát, a kiírás formátumát majd az idézőjelek után vesszőkkel elválasztva fölsoroljuk a változók neveit. A gyakrabban használt karaktereket a következő táblázatban soroltuk föl.:

%d	decimális egész
%u	előjel nélküli decimális egész
%f	lebegőpontos
%c	karakter
%s	sztirng vagy karakter tömb
%e	dupla valós

Lehetőség van arra is, hogy meghatározzuk a változó értéke által elfoglalt mező szélességét. Nézük a következő példákat

```
int v1;
printf("…%5d",v1);
```

A v1 egész változó 5 karakter helyen jelenik meg.

```
float f1;
printf("…%5.2f",f1);
```

Az f valós változót 5 mezőre írja ki 2 tizedes pontossággal.

```
int v1,w=10;
printf("…%*d",w,v1);
```

A * jelentése eben az esetben az, hogy a mezőszélességet az idézőjel utáni első változó határozza meg. Tehát a fenti példában a v1 egész változó 10 szélességű mezőre kerül kiírásra.

Ha egy változó karakter típusú, akkor értékét egyszeres idézőjelek között kell megadnunk.

```
char betu;
betu='A';
```

Egész típusú változónak adhatunk 16-os vagy 8-as számrendszerbeli értéket is.

```
int okt, hex;
okt = 0567;
hex = 0x2ab4;
hex = 0X2AB4;
```

Minden változó esetén figyeljünk a kezdőérték megadására. Ha ezt nem tesszük meg, a változónak akkor is lesz kezdőértéke, de biztosan nem olyan, amit mi szerettünk volna adni.

2.3.1 Felhasználói típus

Ha egy típus neve túl hosszú és gyakran használnunk kell a program során, akkor érdemes egy szinonímával hivatkozni rá: Ennek a módja:

```
typedef típus típusnév
```

Egy konkrét példán keresztül

```
typedef unsigned long int egesz;
egesz n;
```

Néhány további formázási lehetőség a printf()-ben:

\n	új sor
\t	tabulátor
\r	kocsi vissza
\f	soremelés
\v	függőleges tabulátor

2.4 Konstansok

A konstansok megadásának két módját ismertetjük az alábbiakban.

2.4.1 A const kulcsszó hazsnálatával

```
const int a=30;
```

Ebben az esetben vigyázni kell arra, hogy a konstanst inicializáljuk is. Hiba jelzést ad a következő deklaráció:

```
const int a;
a=30;
```

Az így létrehozott konstansok értéke közvetlenül nem változtatható meg. A konstansok, azonban a memórában tárolódnak, így értékük közvetetten mutatók használatával módosítható

2.4.2 Az előfordítónak szóló helyettesítéssel

Az előfordítónak különböző definíciókat, leírásokat adhatunk, erről a későbbikeben még részletesen lesz szó. Most egyetlen példát nézzünk a konstansok megadására

```
#define ADO_KULCS 0.25
```

Az így megadott konstansok a program listájának elején szerepelnek a `#include` beillesztések után. Szintaktikailag a # a sor első karaktere kell hogy legyen, az ilyen sorokat nem zárhatjuk pontosvesszővel, és minden sorban csak egy `#define` állhat. Mivel ezek a leírások az előfordítónak szólnak, ezért minden olyan helyen, ahol a programlistában az `ADO_KULCS` azonosító szerepel, az előfordító 0.25 értéket fog beírni. Ezért az így létrehozott konstansok értéke még indirekt módon sem változtatható.

2.5 Operátorok

A programok írása során gyakran van szükségünk kifejezések felépítésére, változónak történő értékadásra, számolási műveletekre, függvényhívásokra. A C nyelvben ezek a kifejezések oparandusok, függvényhívások és operátorok kombinációjából épülnek fel.

Az operátorokat többféle szempont szerint lehet csoportosítani.

- Az operandusok száma szerint. (egy, kettő, három operandus)
- Az operátor típusa szerint (aritmetikai, logikai, léptető, bitművelet, értékadó, feltételes)
- Az operátor helye szerint (prefix, postfix)

Itt az operátorokat a típusuk szerint tárgyaljuk, de említést teszünk a másik két szempontról is.

2.5.1 Aritmetikai operátorok

A négy alapművelet, a szokásos szimbólumokkal, valamint a maradékos osztás % jellel. A művelet az osztás maradékát adja vissza, természetesen csak egész típusú változók használata esetén alkalmazható. A másik négy művelet mind egész, mind valós operandusok esetén is működik. Vigyázzunk a következőhöz hasonló esetekben.

```
int a=12,b=5;
float f;
f=a/b;
```

A programrészlet után az f értéke a 2.000000 valós érték lesz. Tehát az eredmény típusát az operandusok típusa döntötte el.

A műveletek precedenciája a matematikában megszokott.

2.5.2 Összehasonlító és logikai operátorok

Feltételekben és ciklusokban gyakran kell összehasonlítani különböző értékeket, ennek elvégzésére a hasonlító operátorokat használjuk. Ezek a következők: `<`, `>`, `<=`, `>=`, `==`, `!=`. Ha ezekkel két változót vagy kifejezést hasonlítunk össze, akkor az eredmény **int** típusú lesz, és értéke 1, ha a reláció igaz, illetve 0, ha hamis.

A logikai kfejezésekben gyakran összetett feltételeket is meg kell fogalmazni, erre szolgálnak a logikai operátorok. Ezek a következők: **!** a tagadás művelete, egyoperandusú. **&&** logikai és, **||** logikai vagy műveletek. A műveletek precedenciája a táblázatban.

2.5.3 Léptető operátorok

A változó értékének eggyel való növelésére, vagy csökkentésére szolgálnak. Egyoperandusú műveletek. Postfix és prefix alakban is írhatók. ++ eggyel növeli, -- eggyel csökkenti a változó értékét. Ha egy kifejezésben csak egy változó szerepel, akkor mindegy, hogy postfixes, vagy prefixes alakját használjuk az operátoroknak. (a++ egyenértékű a ++a-val) Ha azonban egy kifejezés kiértékelésében használjuk, akkor már óvatosabban kell bánni a két alakkal.

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

A programrészletben az első értékadás előtt az a értéke 1-gyel nő 5 lesz, ezt kapja az x változó, tehát annak értéke is 5 lesz, a második értékadásban az y megkapja a pillanatnyi értékét az ötöt, majd az a értéke 1-gyel nő, azaz 6 lesz. Az operátorok mind egész, mind valós operátorokkal működnek.

C-ben nem szokás az `a=a+1` értékadás, helyette minden esetben a léptetést használjuk.

2.5.4 Bitműveletek

A műveletek operandusai csak **char**, **short**, **int** és **long** típusú előjel nélküli egészek lehetnek.

A műveletek első csoportja két operandusú. ~ 1-es komplemens, & bitenkénti és, | bitenkénti vagy, ^ bitenkénti kizáró VAGY. Ezeket a műveleteket leggyakrabban *maszkolásra*, vagy bitek törlésére szoktuk használni. A kizáró VAGY érdekes tulajdonsága, hogy ha ugyanazt a maszkot kétszer alkalmazzuk egy értékre, akkor visszakapjuk az eredeti értéket.

A műveletek másik csoportjába a biteltoló műveletek tartoznak. << eltolás balra, >> eltolás jobbra. A felszabaduló pozíciókba 0 kerül, a kilépő bitek elvesznek. A műveletek két operandusúak. `a<<2` az a változó bitjeit 2-vel tolja balra. Nyilvánvalóan az n bittel való balra tolás 2^n-nel való szorzást, míg az n bittel való jobbra tolás 2^n-nel való egészosztást eredményez.

2.5.5 Értékadó operátorok

Az értékadás történhet a más nyelvekben megszokottak szerint.

```
a=érték, vagy a=kifejezés formában
```

Van azonban olyan forma is, mely a hagyományos nyelvektől teljesen idegen.

```
b=2*(a=4)+5
```

Ebben az esetben az a és a b változó is kap értéket. Az értékadás operátor mindig jobbról balra értékelődik ki, tehát a kiértékelés után a fenti kifejezésben a értéke 4, b értéke pedig 13 lesz. A kiértékeklés ilyen sorrendje miatt van az, hogy a C-ben az összetett értékadás is működik.

`a=b=c=0` értékadás után mindhárom változó értéke 0 lesz.

Van az értékadásnak C-ben egy tömörebb formája is. Általános alakban a következőképpen írható le:

`változó=változó op kifejezés` helyett a `változó op=kifejezés`

Ez a forma általában gyorsabb kódot és áttekinthetőbb listát eredményez. A C programokban természetesen mindkettő forma használható, de igyekezzünk a másodikat előnyben részesíteni. A mellékelt táblázatban összefoglaljuk ezeket a rövidített értékadásokat.

Hagyományos forma	Tömör forma
a=a+b	a+=b
a=a-b	a-=b
a=a*b	a*=b
a=a/b	a/=b
a=a%b	a%=b
a=a<<b	a<<=b
a=a>>b	a>>=b
a=a&b	a&=b
a=a|b	a|=b
a=a^b	a^=b

2.5.6 Feltételes operátor

A C-ben ez az egyetlen operátor, melynek három operandusa van. Általános alakja a következő:

```
kifejezés1 ? kifejezés2 : kifejezés3
```

Itt először a kifejezés1 értékelődik ki, ha ennek értéke nem 0, azaz IGAZ, akkor a kifejezés2 adja a feltételes kifejezés értékét, különben pedig a kifejezés3. A feltételes kifejezés típusa mindig a kifejezés2 és kifejezés3 típus közül a nagyobb pontosságú típusával egyezik meg.

A következő példában c értéke a és b közül a kisebbel lesz egyenlő.

```
c = a < b ? a : b
```

2.5.7 Pointer operátorok

A mutatókról az eddigiek során még nem volt szó, de a teljesség kedvéért megemlítünk két egyoperandusú műveletet, mely a mutatókhoz (pointerek) kötődik. & a címe operátor.

```
int a , *ptr;
ptr = &a
```

Ebben a példában a ptr egy egész típusú változóra mutat, értékadásnál pedig az a változó memóriabeli címét kapja meg. Ha erre a címre új értéket akarunk írni, akkor a * (indirekt hivatkozás) operátort kell használnunk.

```
*ptr = 5 egyenértékű az a=5 értékadással.
```

2.5.8 Típuskonverziók

A C-ben kétfajta típuskonverzió létezik, az implicit (automatikus) és az explicit. Az első a C nyelvbe rögzített szabályok szerint történik a programozó beavatkozása nélkül, a második pedig a típuskonverziós operátor segítségével. Ennek általános alakja:

```
(típusnév) kifejezés
```

Az implicit konverzióval kapcsolatban elmondhatjuk, hogy általában a szűkebb operandus információvszteség nélkül konvertálódik a szélesebb operandus típusára.

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

Ebben az esetben az i float-ra konvertálódik.

```
j=i+f;
```

Itt viszont vigyáznunk kell, mert adatvesztés lép fel, az összeg törtrésze elveszik.

Explicit konverziót kell végrehajtanunk a következő példában, ha f-be nemcsak az egész osztás hányadosát szeretnénk betenni

```
int a = 12,b = 5;
float f;
f = (float) a / (float) b;
```

A következő táblázatban összefoglaltuk az említett műveleteket precedenciájuk szerint rendezve. Valamint a kiértékelés sorrendjét is megadtuk. A kiértékelés sorrendje akkor kerül előtérbe, ha egy kifejezésben egyenlő precedenciájú operátorok szerepelnek zárójelezés nélkül

Operátor	Kiértékelés sorrendje
`! ~ - ++ -- & * (típus)`	Jobbról balra
`* / %`	Balról jobbra
`+ -`	Balról jobbra
`<< >>`	Balról jobbra
`< <= > >=`	Balról jobbra
`== !=`	Balról jobbra
`&`	Balról jobbra
`^`	Balról jobbra
`|`	Balról jobbra
`&&`	Balról jobbra
`||`	Balról jobbra
`?:`	Jobbról balra
`= += -= *= /= %= <<= >>= &= |= ^=`	Jobbról balra

2.6 Adatok beolvasása a billentyűzetről

A formázott adatbeolvasást a `scanf` függvény segítségével tehetjük meg. A függvény általános formája a következő:

```
scanf(formátum, argumentumlista)
```

A scanf karaktereket olvas a billentyűzetről, majd a formátum alapján értelmezi azokat, ha a beolvasott karakterek megfelelők, akkor konvertálja őket. Ha az input valamilyen ok miatt nem felel meg a formátum előírásainak, akkor a scanf befejezi az olvasást, még akkor is, ha az argumentumlista szerint további karaktereket is be kellene olvasnia. A scanf függvénynek visszatérési értéke is van. A sikeresen beolvasott adatok számát adja vissza. Nézzünk néhny példát a scanf használatára.

```
int a ;
char c;
printf(”Kérek egy egész számot és egy betűt”);
scanf(”%d%c”,&a,&b)
```

A példából látszik, hogy az egyszerű adatokat cím szerint kell beolvasni. Az argumentumlistában az &a és az &c a változók memóriabeli címére utal. A formátumban ugyanazokat a karaktereket használhatjuk, mint a printf esetében korábban tettük. Ez a sor egy számot és egy karaktert olvas be egymás után, nem tesz közéjük semmilyen elválasztó jelet. Nyilván, ha egy scanf-fel több értéket is akarunk beolvasni, akkor valamilyen határolóra szükség van.

```
int a ;
char c;
printf(”Kérek egy egész számot és egy betűt vesszővel elválasztva”);
scanf(”%d,%c”,&a,&b);
```

Figyeljük meg a változtatást. A formátumban egy vesszőt tettünk a második % jel elé. Ilyenkor a scanf beolvassa a vesszőt is, de azt nem tárolja. Ilyen módon bármilyen határoló karaktereket előírhatunk beolvasáskor.

A scanf segítségével sztringeket is olvashatunk be. Ebben az esetben nem használjuk az & operátort.

```
char sz[30];
scanf(”%s”,sz);
```

A scanf egy hasznos lehetősége, hogy az adatok szűrését is lehetővé teszi. Az előbbi deklaráció szerinti sz változó értéke csak számjegy, vagy hexadecimális jegy lehet, akkor azt a következő szűréssel valósíthatjuk meg:

```
scanf(”%[0-9a-fA-F]”,sz)
```

A komplementer halmaz megadására is van módunk:

```
scanf(”%[^0-9]”,sz)
```

Ezzel a szűréssel csak betűk kerülhetnek a sztringbe

2.7 Iterációk

Mint minden magas szintű programozási nyelvben a C-ben is vannak olyan utasítások, melyek egy feltételtől függően többször is végrhajtják ugyanazt az utasítást, vagy utasítás blokkot. A következőkben ahol utasítást írunk helyettesíthető utasításblokkal is. Utasításblokk:

```
{
  utasítás1;
  utasítás2;
  …
  utasításn;
}
```

2.7.1 while ciklus

A while ciklus általános alakja:

```
while (kifejezés)
  Utasítás
```

A ciklusmag utasításai addig hajtódnak végre, amíg a kifejezés értéke nem nulla. (Ez a logikai igaznak felel meg, tehát, ha a kifejezés egy logikai kifejezés, akkor itt a ciklusba lépés feltételét adjuk meg)

Nézzünk egy példát a while ciklusra. Adjuk össze 1-től n-ig a természetes számokat!

```
#include<stdio.h>

main()
   {
      long osszeg=0;
      int i=1,n=2000;

      printf(”Az első %d egész szám összege: ”,n);
      while (i<=n)
         {
            osszeg+=i;
            i++;
         }
      printf(”%ld”,osszeg);
   }
```

A printf-ben látható `%ld`-ben az `l` hosszú (`long`) egészre utal.

A while ciklust gyakran szoktuk használni arra, hogy egy bizonyos billentyű leütésére várakozzunk.

```
while((ch=getch()) !=27);
```

Az ESC billentyű leütésére vár

2.7.2 for ciklus

A for ciklust a leggyakrabban akkor használjuk, ha előre tudjuk, hogy egy utasítást hányszor akarunk végrehajtani. Az utasítás általános alakja egyszerű formában a következő lehet:

```
for ( kifejezés1;kifejezés2;kifejezés3)
  Utasítás
```

A kifejezés1-ben állítjuk be a ciklusváltozó kezdő értékét, a kifejezés kettőben a ciklusba való lépés feltételét, ez a leggyakrabban egy logikai kifejezés, a kifejezés3-ban pedig léptetjük a ciklusváltozót.

Példaként írjuk át az előző programot for ciklust használva.

```
#include<stdio.h>

main()
   {
      long osszeg;
      int i=1,n=2000;

      for (i=1,osszeg=0;i<=n;i++)
         osszeg+=i;
      printf(”Az első %d szám összege: %ld”,n,osszeg);
   }
```

Figyeljük meg a for ciklus fejében a , operátort. Ezt a korábbiakban nem említettük, szerepe az hogy egy utasításban több kifejezést is elhelyezhetünk. Itt az `osszeg=0` értékadás még a `kifejezés1` része. Mivel a ciklus magja összesen egy utasítást tartalmaz, ezért a fenti for ciklus lényegesen rövidebben is leírható:

```
for (i=1,osszeg=0;i<=n;osszeg+=i++);
```

Itt feltétlenül kell egy ; az utasítás végére, ezzel jelezzük, hogy a for ciklus csak egy üres utasítást tartalmaz. Vigyázzunk azonban, ha egy egyéb uatsításokat tartalmazó for ciklusban erre a helyre ;-t teszünk meglepődve tapasztalhatjuk, hogy a ciklusmag többi utasítása nem fog végrehajtódni.

2.7.3 do-while ciklus

Ezt a ciklust igen ritkán használjuk. Minden programozási feladat megoldható az előző két ciklus alkalmazásával, van azonban néhány olyan feladat, mely rövidebb kódot eredményez, ha a do-while ciklust használjuk. (Pl. a bináris keresés) A ciklus általános alakja:

```
do
  Utasítás
while (kifejezés)
```

A ciklusba itt is addig lépünk, amíg a kifejezés értéke nem 0, logikai kifejezés esetén amíg a kefejezés igaz. Alapvetően abban különbözik az előző két ciklustól, hogy itt a ciklusmag utasítása legalább egyszer végrehajtódik.

Nézzük a következő példát a do-while ciklusra. Bekérünk egy egész számot és kiírjuk a fordítottját.

```
#include<stdio.h>

main()
  {
    int szam, jegy;

    printf(”Kérek egy egész számot:”);
    scanf(”%d”,&szam);
    printf(”\nA fordítottja: ”);

    do
      {
        jegy = szam % 10;
        printf(”%d”,jegy);
        szam /= 10;
      } while ( szam != 0);
  }
```

2.8 Szelekciók

A C nyelvben három elágazás típust használhatunk. Az egyágú (if szerkezet), a kétágú (if-else szerkezet) és a többágú (switch szerkezet)

2.8.1 if utasítás

Az utasítás általános alakja:

```
if (kifejezés)
  utasítás
```

Itt az `utasítás` csak akkor hajtódik végre, ha a kifejezés nem nulla. (IGAZ)

Példaként kérjünk be egy valós értéket, de csak az]1,10[intervallumba eső értéket fogadjuk el jó értéknek.

```
#include<stdio.h>

main()
  {
    float v;
    printf(”Kérek egy valós számot\n”);
    printf(”1 és 10 között: ”);
    scanf(”%f”,v);
```

```
    if (v>1 && v< 10)
      printf(”Jó érték!!”);

  }
```

2.8.2 if-else szerkezet

Az utasítás általános alakja:

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

Ha a kifejezés értéke nem nulla (IGAZ), akkor az `utasítás1` hajtódik végre, ha pedig nulla (HAMIS) az `utasítás2`.

Az előző példánál maradva, ha az érték rossz, akkor írassuk ezt ki az else ágon!

```
  else
    printf(”Az érték rossz”);
```

Ezt a két sort az előző programban az üres sor helyére kell beszúrni. Vigyázzunk azonban, ebben az esetben a Jó érték sorának végén nem állhat ; ugyanis ekkor egy else kulcsszóval találkozna a fordító, amivel viszont nem kezdődhet parancs.

2.8.3 switch utasítás

Az utasítás több irányú elágazást tesz lehetővé, de csak abban az esetben, ha egy egész kifejezés értékét több konstanssal kell összehasonlítani. Általános alakja:

```
switch (kifejezés)
  {
    case konstans1:
      utasítás1;
    case konstans2:
      utasítás2;
    ….
    default:
      utasítás;
  }
```

A switch utasítással nagyon átgondoltan kell bánni. *Általános esetben ha az egyik case-nél találunk egy belépési pontot, akkor az utána következő case cimkék után álló utasítások is végre fognak hajtódni.* Hacsak nem pontosan ez a szándékunk, akkor minden utasítást `break`-kel kell zárni, melynek hatására a vezérlés a switch utáni első utasításra kerül. Általában a `default` eseteket is `break`-kel szoktuk zárni, mert ez nem feltétlenül az egyes esetek végén áll, bárhol elhelyezhető a szerkezetben. Ha ugyanazt az utasítást akarjuk végrehajtani több konstans érték esetén is, akkor a konstansokat egymástól :-tal elválaszva soroljuk fel.

Nézzünk egy rövid példaprogramot a switch illusztrálására:

```
#include <stdio.h>

main()
{
  int menu, n1, n2, t;
```

```
    printf("Írjon be két számot: ");
    scanf("%d %d", &n1, &n2 );
    printf("\n Válasszon\n");
    printf("1=Összeadás\n");
    printf("2=Kivonás\n");
    scanf("%d", &menu );
    switch( menu ) {
      case 1: t = n1 + n2; break;
      case 2: t = n1 - n2; break;
      default: printf("Nem jó választás\n");
    }
    if( menu == 1 )
      printf("%d + %d = %d\n", n1, n2, t );
    else if( menu == 2 )
      printf("%d - %d = %d\n", n1, n2, t );
  }
```

Figyeljük meg, hogy minden választás után használtuk a break-et. A default után nem használtuk, de ha a switch elején van, akkor ott is érdemes odafigyelni rá.

2.9 Tömbök

Az eddigiek során olyan változókat használtunk csak, melyek egy érték tárolására voltak alakalmasak. Gyakran van azonban szükségünk arra, hogy ugyanazzal a változónévvel több értékre is tudjunk hivatkozni. Ebben az esetben haszáljuk a tömböket. (Egy osztály minden tanulójához hozzárendeljük a magasságát, ilyenkor nyilván érdemes változónévként az osztály azonosítóját használni, a tanulókra pedig a napló sorszámával)

2.9.1 Egydimenziós tömbök

A deklaráció szintaxisa:

```
Típus tömbnév[méret]
```

Konkrét példák:

```
int a[10]
char betuk[5]
```

Hivatkozások a Pascalban is megszokott módon történhetnek:

```
betuk[1]='C'; a[2]=23;
```

A C-ben a tömb elemeinek sorszámozása minden esetben 0-tól indul, azaz a fenti példákban vigyázni kell, mert nem hivatkozhatunk a[10]-re, mert az már a tömb 11. eleme lenne.

A tömböknek már a létrehozáskor is adhatunk kezdőértéket. Ennek szintaxisa a következő lehet:

```
int a[5]={1,2,3,4,5}
int b[]={2,5,7,8,11,22,33}
```

Az első esetben megmondtuk a tömb méretét és ennyi darab elemet is soroltunk fel, ha több elemet adtunk volna meg, akkor a fordító hibajelzéssel leáll, ha kevesebbet, akkor pedig a változó *tárolási osztályától*[3] függően vagy 0, vagy határozatlan értékű lesz. A második esetben nem adtuk meg az elemek számát a szögletes zárójelben, csak felsorolással, ilyenkor a tömb pontosan 7 elemű lesz.

[3] Lényegében a változó elérhetőségét szabályozza, kicsit hasonló a Pascal globális és lokális változóihoz, a későbbiekben még lesz róla szó.

A tömb méretét akár futásidőben is meg tudjuk mondani. Ha egy program tesztelése során változtatjuk egy tömb méretét (új elemeket veszünk fel a fölsorolásba), akkor érdemes a definíciós részben a következő sor beszúrni:

```
#DEFINE MERET (sizeof(b)/sizeof(b[0]))
```

Példaként nézzük meg hogyan lehet egy 10 hosszúságú vektor elemeit beolvasni, és összegezni:

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

main()
{
  int a[10];
  int s=0,i;

  for (i=0; i<SZAM;i++)
  {
    prinf(”a[%d]=”,i);
    scanf(”%d”,&a[i]);
    s+=a[i];
  }
  printf(”\n Az összeg: %d”,s);
}
```

2.9.2 Sztringek

A C-ben nincs külön sztring típus, ezért az egy dimenziós tömböket gyakran használjuk sztringek definiálására. Tulajdonképpen a sztring egy egydimenziós karaktertömb lesz. A sztring végét egy \0 karakter fogja jelezni, ezért ha egy 10 hosszúságú sztringet szeretnénk kezelni, akkor azt feltétlenül 11 hosszúságúra kell létrehoznunk.

```
char sztring[11];
char sz[]={’C’-,’n’,’y’,’e’,’l’,’v’,’\0’);
```

Értékadás helyett célszerűbb és biztonságosabb a következő megadási mód:

```
char sz[]=”C-nyelv”;
```

Ebben az esetben ugyanis a sztring végét jelző `\0` karaktert a fordító teszi ki a sztring végére.

A következő táblázatban néhány sztingkezelő függvényt sorolunk föl.

Név	Leírás	Példa
strcat(sz1,sz2)	Az sz1 sztringhez fűzi az sz2 sztringet	sz1=”hello”; sz2=”world” strcat(sz1,sz2); printf(” %s\n”,sz1)
		Eredménye: helloworld
strcpy(sz1,sz2)	Az sz1 stringbe másolja az sz2 sztringet	sz2=”world” strcat(sz1,sz2); printf(” %s\n”,sz1)
		Eredménye: world
strcmp(sz1,sz2)	Összehasonlítja a két sztringet, ha egyenlők, akkor , ha nem egyenlők,	sz1=”Hello”; sz2=”HeLlO” n=strcmp(sz1,sz2); printf(”%d\n”,n)

	akkor nem 0 értékkel tér vissza, különben igen.	**Eredménye: 32**
strcmpi(sz1,sz2)	Ugyanaz, mint az előző, csak a kis és a nagy betűk között nem tesz különbséget.	Az előző adatokkal
		Eredménye: 0

strchr(sz1,c)	Megnézi, hogy az sz1 stringben először hol fordul elő a c karakter	char c='e', sz1[]="Hello", *ch;
		ch=strchr(sz1,c);
		printf("%d\n", ch-sz1);

		Eredménye: 1[4]
strrchr(sz1,c)	Ugyanaz, mint az előző, csak jobbról indul	
strlwr(sz), strupr(sz)	A sztringet kis, illetve nagy betűs formátumúvá alakítja	sz="hello" printf("%s\n",strupr(sz));
		Eredménye: HELLO

strncpy(sz1,sz2,n)	Az sz1 sztringbe másol sz2-ből n db karaktert;	sz2="Hello";n=2;
		strncpy(sz1,sz2,n);sz[2]="\0"[5];
		printf("%s\n",sz1);

		Eredménye: He

strncat(sz1,sz2,n)	Az sz1 szringhez fűz n db-ot az sz2-ből	sz1="Hello"; sz2="World"; n=2;
		strncat(sz1,sz2,n);
		printf("%s\n",sz1);

		Eredménye: HelloWo
strrev(sz)	Megfordítja a sztringet	sz1="Hello"; printf("%s\n",strrev(sz))
		Eredménye:olleH
strlen(sz)	Egy sztring hosszát adja meg	n=strlen(sz);

Ha ezeket a sztringkezelő függvényeket használni akajuk,akkor mindenképpen szükség van a `#include <string.h>` beszúrásra.

Nézzünk egy példaprogramot arra, hogyan lehet egy sztringet karakterenént végigolvasni anélkül, hogy tudnánk a hosszát:

```
#include <stdio.h>

main()
{
   int sz[]="C programozási nyelv";
   int i=0;
```

[4] Figyeljük meg, hogy az eredmény 1, ugyanis a sorszámozás most is 0-val kezdődik. A változó deklarációkat is figyeljük meg!

[5] Figyeljünk föl rá, hogy a sztring végére nekünk kell kitennünk a végét jelző karaktert, különben nem fog helyesen működni!

```
  while (s[i])
  {
    printf(”\n%c”,s[i];
    i++;
  }
}
```

2.9.3 Kettő és több dimenziós tömbök

A C nyelvben is lehetőségünk van kettő vagy több dimenziós tömbök használatára. Deklarálni a következőképpen kell:

```
típus név [méret1] [méret2] …;
```

Konkrét pédaként egy valós értékeket tatalmazó két dimenziós tömbre:

```
float f [5][2];
```

Az első index ebben az esteben a sor index, a második pedig az oszlop index. A hivatkozás ennek megfelelően:

```
f[1][2]=4.73;
```

Lehetőség van itt is a kezdőérték megadására:

```
int matrix[3][2]={ {1,2}, {3,4}, {5,6}};
```

Nagyon fontos, hogy az indexek itt is minden esetben 0-tól indulnak!

Egy két dimenziós tömb táblázatos megjelenítése:

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

#define SOR 4
#define OSZLOP 3

main()
{
  int a[SOR][OSZLOP];
  int i,j;

  clrscr();
  randomize();
  for (i=0;i<SOR;i++)
    for (j=0;j<OSZLOP;j++)
      a[i][j]=rand()%100;
  for (i=0;i<SOR;i++)
  {
    for (j=0;j<OSZLOP;j++)
      printf(”%d\t”,a[i][j]);
    printf(”\n”);
  }
}
```

2.10 Felhasználó által definiált típusok

Az előző részben említett összetett adattípust tömböknek neveztük, a tömbök minden eleme azonos típusú kell hogy legyen. Gyakran szükségünk van azonban olyan egymással összetartozó elemek tárolására is, melyek nem azonos típusúak, mégis egy egyedre jellemzők. Ezket leggyakrabban adatbázisokban találjuk meg. Gondoljunk itt egy személy kereszt és vezetéknevére, születési idejére, alapfizetésére. Ezek különböző típusú adatok mégis ugyanarra a személyre vonatkoznak.

2.10.1 Struktúrák

Egy ilyen adattípust valósítanak meg a struktúrák. Ennek deklarációja általános formában:

```
struct név  {
     típus1 tag1;
     típus2 tag2;
     …….
     }
```

A bevezetőben említett konkrét példban leírt struktúra:

```
struct szemely{
       char vnev[20];
       char knev[15];
       int szev;
       float fizetes;
       }
```

Ha ilyen típusú változót akarunk létrehozni, akkor annak módja:

```
struct szemely sz;
```

Erre a változóra a nevével és a struktúrán belüli tag nevének megadásával hivatkozhatunk a .(pont) operátor segítségével:

```
sz.vnev=”Kovács”;
```

A struktúrát atypedef-fel együtt is használhatjuk. Ilyenkor a `struct` utáni azonosító el is maradhat, és csak utána kerül a a típust azonosító név:

```
typedef struct  {
       char vnev[20];
       char knev[15];
       int szev;
       float fizetes;
       } szemely
```

Ebben az estben természetesen a változó típusának megadásakor is hiányzik a `struct`.

```
szemely sz;
```

2.10.2 Struktúrát tartalmazó tömbök

Ha egy struktúra típusú változót létrehoztunk, akkor annak segítségével csak egyetlen egyed jellemzőit tudjuk tárolni. Mit tehetünk, ha több egyedről is ugyanazokat a jellemzőket szeretnénk raktározni? Kézenfekvő megoldás olyan tömbök alkalmazása, melynek minden egyes eleme az adott struktúra típusú.

Az előző példánál maradva egy vállalatnak maximálisan 20 dolgozójáról a fenti adatokat szeretnénk tárolni, akkor a

```
szemely sz[20];
```

adatszerkezettel dolgozhatunk. Az adatszerkezet egyes elemeire való hivatkozáskor a struktúra és a tömb hivatkozásokat vegyesen alkalmazzuk.

```
sz[2].szev=1961;
```

2.11 Függvények

Függvényeket a következő esetekben szokás írni:

- Ha ugyanazt a tevékenységsorozatot többször is el kell végeznünk ugyanolyan típusú, de más-más értéket fölvevő változókkal.
- Ha a programunkat struktúráltan, jól olvashatóan szeretnénk megírni. Ez nagyon afontos a későbbi módosíthatóság miatt.
- Természetesen az első kettő eset együtt is fennállhat.

A függvény defníció általános alakja:

```
visszatérési_érték_típus fvnév (típus1 vált1,típus2 vált2,……)
{
  a függvény teste
  return v1;
}
```

Konkrét példa egy egyszerű függvényre:

```
int osszeg(int a, int b)
{
  int s=a+b;
  return s;
}
```

Ha egy függvénynek nincs visszatérési értéke, akkor a `void` kulcsszót használjuk:

```
void fnev(típus1 vált1,…)
```

2.11.1 Paraméterátadás

A paraméterátadás a Pascal-hoz hasonlóan itt is történhet cím szerint és értékszerint.Az elve szintén ugyanaz. Az érték szerinti paraméterátadásra az iménti függvény lehet egy példa. Címszerinti paraméterátadásnál azonban már sokkal jobban oda kell figyelni, mint a Pascalban.

Példaként írjunk egy függvényt, mely a paraméterben megadott változókat fölcseréli.

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

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

main()
{
  int k=2;l=4;
  printf(”%d,%d”,k,l);
```

```
  csere(&k,&l);
  printf(”\n%d,%d”,k,l);
}
```

Nézzük meg figyelmesen a listát! Már rögtön a függvény fejében észrevehetünk egy változást, a *-ot használjuk, ezzel a változó memóriabeli helyén található értékre utalunk. A másik változás a függvényhíváskor figyelhető meg, az C opertort használjuk, ezzel jelezzük azt, hogy nem a változó értékét, hanem e memóriabeli címét adjuk át. Nyilván a függvényben emiatt kell használnunk a * operátort. Ezek az operátorok a mutatókhoz (pointer) kötődnek, a későbbiekben még lesz róluk szó.

Vegyük észre azt is, hogy a függvénynek nincs visszatérési értéke, annak ellenére, hogy az átadott változók értéke megváltozott.

2.11.2 Lokális és globális változók

A lokális változókat csak azok a ***blokkok***[6] látják, ahol deklarálva lettek, ez alól a main sem kivétel. Ezek a változók a veremben tárolódnak, tehát minden függvényhíváskor törlődik az értékük.

A globális változókat minden függvény látja, értéküket módosíthatja. Ezeket a változókat minden függvény törzsén kívül kell deklarálni, célszerű rögtön a `#define` után megtenni ezt. Ajánlott a globális változókat egy tömbben deklarálni, lehetőség van ugyan arra, hogy két függvény között is deklaráljunk változót, ebben az esetben viszont az előbb lévő függvény nem látja az alatta lévő változót, annak ellenére sem, hogy az függvényeken kívül lett létrehozva.

2.11.3 Automatikus és statikus változók

Bevezetésként tanulmányozzuk a következő programot:

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

void demo()
{
      static int sv=0;
      auto int av=0;

      prtintf("sv= %d, av= %d",sv,av);
      ++sv;
      ++av;
}

main()
{
      char c;
      int i=0;

      clrscr();
      while (i<3)
      {
            demo();
            i++;
      }

      while ((c=getch()) !=13);

}
```

[6] Nem véletlen a kiemelés. Itt valóban blokkról van szó, azaz lehet olyan lokális változóm is, amit csak egy for ciklus végrehajtási része lát, a függvény többi része nem. Ez egy jelentős eltérés a C és aPascal lokális változói között.

A program kimenete:

```
sv=0, av=0;
sv=1; av=0;
sv=2; av=0
```

A statikus változókat csak egyszer hozza létre a fordító és értéket is csak egyszer kapnak, a legelső függvényhíváskor. Statikus változót lokálisan és globálisan is létrehozhatunk, értékét a program futása során végig megtartja, a különbség a kettő között csak a láthatóságban van. A lokálisan létrehozott statikus változó csak a blokkján belül látható. A fenti program egy lokális satikus változót mutatott be.

Az automatikus változók minden függvényhíváskor újra deklarálódnak, és mindig fölveszik a megadott kezdőértéket. Ha nem írjuk ki ezeket a módosító jelzőket, akkor a változó autmatikus lesz. Az automatikus változók dinamikusan tárolódnak a veremben.

2.11.4 Register módosító jelző

A register kulcsszó azt jelenti a fordító számára, hogy az adott változóhoz gyorsan szeretnénk hozzáférni. Arra utasítjuk tehát, hogyha módjában áll, akkor ne a memóriában, hanem regiszterekben tárolja. Nyilván akkor folyamodunk ilyen technikához, ha úgy gondoljuk,hogy az adott változót gyakran módosítjuk. Regiszterbe char, int és pointer típust tehetünk az esetek többségében.

Arra semmi garancia nincs, hogy a fordító tudja teljesíteni kérésünket, de ha tudja, akkor az eredmény gyorsabb futás lesz.

```
void csere(register int *a, register int *b)
{
  register int s;
  s=*a;
  *a=*b;
  *b=s;
}
```

2.11.5 Vektor átadása függvénynek

A következő példa bemutatja, hogy egy vektort hogyan adhatunk át paraméterként egy függvénynek. A függvény az átadott vektor legnagyobb elemével tér vissza.

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

int maximum(int a[], int n)
{
      int i;
      int max=a[0];

      for (i=1;i<n;i++)
            if (max<a[i]) max=a[i];

      return max;

}

main()
{
      char c;
      int b[]={12,5,6,3,9,11,13,21,10,1,34};

      clrscr();
```

```
	printf("\n a b[] maximuma: %d",maximum(b,sizeof(b)/sizeof(b[0])));

	while ((c=getch()) !=13);

}
```

A programot vizsgálva adóthat az az ötletünk, hogy a tömb méretét fölösleges volt átadni a függvénynek, azonban ez sajnos nem igaz. A függvény fejében lévő `int a[0]` ugyanis csak annyit közöl a fordítóval, hogy egy egészeket tartalmazó vektor fog jönni paraméterként.

2.12 Mutatók (Pointerek)

A C programozási nyelvben van egy különös adat típus: a mutató. Ez egy változónak a memóriabeli címét veheti föl értékül. Definiálás során a mutató típusú változó neve előtt a * jelet kell használnunk. (Ezt nevezik inderektség operátornak is) Egy mutató értéket az & (címe) operátorral kaphat. A pointer által mutatott címet szintén a * operátorral kaphatjuk vissza. Nézzünk erre egy konkrét példát:

```
int a=3,b, *p;/* a pegy egész típusú változóra mutató pointer */
p=&a;    /* p az a címét veszi föl értékként */
b=*p;    /* b megkapja a p által mutatott címen lévő értéket, ebben az
esetben 3 lesz */
```

A következő programban mi fog megjelenni a két kiírás után?

```
#include <stdio.h>

main()
{
	char c;

	int a=10,b=15, *p;
	clrscr();

	p=&a;
	*p+=a+b;
	printf("%d",a);

	p=&b;
	*p+=a+b;
	printf("%d",b);

	while ((c=getch()) !=13);
}
```

A mutatóknak nem ez a bűvészkedés adja a jelentőségét. Segítségükkel dinamikusan kezelhetjük a memóriát, mindig csak annyit használva, amennyire az adatoknak éppen szüksége van.

2.12.1 Vektorok és mutatók

A C programozási nyelvben igen szoros kapcsolat van a a vektorok és az egyszeres indirektségű[7] mutatók között. Ez a vektorok tárolási módjából ered. A vektorok a memóriában sorfolytonosan helyezkednek el. Ha egy pointert a vektor első elemére irányítunk, akkor a pointer aritmetika szabályai szerint ehhez 1-et hozzáadva a vektor második elemét fogjuk megkapni.[8]

```
int *p,a[10];
p=&a[0];
```

[7] `int **p`; p egy olyan mutató, mely egy egészre mutató mutatóra mutat. Kétszeres indirektség.

[8] Mutatók esetén a `p++` növelés nem eggyel növeli p értékét, hanem a szomszédos címre mutat. Azaz, ha amutató eredetileg egy char típusú változóra mutatott, akkor a növelés 1, ha egy int típusúra, akkor a növelés 2, ha valamilyen összetett struktúrára, akkor pedig ennél sokkal több.

Ekkor a *p hivatkozás a vektor első elemét fogja jelenteni. Teljesen egyenértékű a következő két hivatkozás:

```
*p=5; a[0]=5;
```

Mivel a kapcsolat ilyen szoros a vektor és a vektor első elemére mutató pointer között, ezért a vektoros és a pointeres hivatkozások felcserélhetők. a fenti deklarációk szerint a vektor i-edik elemére való hivatkozások:

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

Az első kettő tömbös, a második kettő pedig pointer típusú hivatkozás. Jól jegyezzük meg tehát, hogy az *a tömbnév és a p mutató is az elemek sorozatának első elemét jelenti*.

Pascal programokban, ha szükségünk volt egy vektorra, akkor azt már változó deklarációban létre kellet hoznunk és a méretét is be kellett állítanunk. Ha a vektor elemszáma elérheti a 100-at is, de az esetek 99%-ában nekünk csak 10 elemre van szükségünk, nem tehettünk mást, mint hogy 100 hosszúságúra hoztuk létre a vektort, ezzel jelentős memória területet lefoglaltunk. A C nyelvű programokban nagyon egyszerűen létrehozhatunk dinamikus helyfoglalású vektorokat. Ezeknél futási időben dől el, hogy milyen hosszúságúak lesznek, ha a további feldolgozáshoz nincs szükség rájuk, akkor az általuk lefoglalt memória felszabadítható.

Nézzünk erre egy példaprogramot. A program megkérdezi, hogy hány elemű vektorral kívánunk dolgozni, majd helyet foglal a memóriában az elemeknek, véletlen számokkal feltölti a vektort, kiírja az elemeket és összegüket, majd a végén felszabadítja a lefoglalt memória területet.

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

main()
{
  char c;

  unsigned int *p,n,i;
  long int osszeg=0;

  clrscr();
  randomize;

  printf("Hány adattal akarsz dolgozni: ");
  scanf("%u",&n);

  /* Helyfoglalás a memóriában, p az első helyre mutat */
  p=(unsigned int *) calloc(n,sizeof(unsigned int));
  if (!p)
    {
      printf("Nincs elég hely a memóriában");
      return -1;
    }

  for (i=0;i<n;i++)
    p[i]=random(5000);       /* Tömbös hivatkozás */

  for (i=0;i<n;i++)
    {
      printf("%u,",*(p+i));   /*  Pointeres */
      osszeg+=*(p+i);         /* hivatkozás */
    }

  printf("\n%ld",osszeg);

  free(p);    /* Felszabadítja lefoglalt memóriát */
```

```
  while ((c=getch()) !=13);

}
```

Magyarázatok a fenti programhoz: Ha a memóriakezelő függvényeket akarjuk használni, akkor szükségünk van az `stdlib.h` beemelésére. A helyfoglalást a memóriában a `calloc`, vagy a `malloc` függvények hívásával végezhetjük el. A `calloc` függvénynél meg kell mondanunk, hogy hány elem számára szertnénk helyet foglalni, és hogy egy elemnek mekkora a mérete, ebben a sorrendben. A függvény a lefoglalt memóriaterületet rögtön feltölti nullával. A `malloc` függvénnyel pedig azt kell közölni, hogy mekkora memóriaterületet szeretnénk lefoglalni (byte-ban).

Ha már nincs szükségünk a lefoglalt területre, akkor ezt a `free` függvénnyel fölszabadíthatjuk.

Figyeljük meg a p-nek történő értékadást! A `calloc` függvény visszatérési értéke egy típus nélküli mutató, ezt egy típuskonverzióval át kellet alakítani a p típusának megfelelő alakra.[9] Minden esetben meg kell vizsglni, hogy sikeres volt-e a helyfoglalás. Ha nincs elég memória, akkor a `calloc` függvény `NULL` (nulla) értékkel tér vissza, ebben az esetben a `return –1` hatására a program kilép a `main()` függvényből és befejezi futását.

A programban a pointeres és tömbös hivatkozás vegyesen lett használva, mutatva ezzel a kettő teljes egyenértékűségét.

2.12.2 Kétdimenziós tömbök és pointerek

Természetesen két dimenziós tömböket is lehet dinamikusan kezelni. Erre három különböző módszert mutatunk be. Mindhárom program csupán annyit csinál, hogy egy mátrixot feltölt véletlen számokkal, majd táblázatos formában megjeleníti. A programok után rövid magyarázatok is lesznek.

2.12.2.1 Dinamikus tömb egy vektorban

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

main()
{
  char c;
  int n,m,*p;
  int i,j;

  clrscr();
  randomize();

  printf("Sorok száma: ");
  scanf("%d",&n);
  printf("Oszlopok száma: ");
  scanf("%d",&m);
  p=(int *) calloc(n*m,sizeof(int));
  if (!p)
    {
      printf("Nincs elég memória!");
      return -1;
    }

  for (i=0;i<n;i++)
    {
      for (j=0;j<m;j++)
        {
```

[9] Ilyen típuskonverzióval találkoztunk már korábban. Nézzük meg a típuskonverziók című fejezetet!

```
            p[i*m+j]=random(10);          /* Tömbös hivatkozás */
            printf("%3d",*(p+i*m+j));     /* Pointeres hivatkozás */
          }
        printf("\n");
      }

   free(p);

   while ((c=getch()) !=13);

}
```

A memórifoglalás ebben a programban valóban futási időben történik. Ha közelebbről is szemügyre vesszük a listát látható, hogy itt a mátrixot valójában egy vektorban tároljuk. Az aktális pozíciót pedig a

hely=aktuális_sor*oszlopszám+aktuális_oszlop

formulával határoztuk meg. Figyeljünk föl itt is a két fajta hivatkozásra!

2.12.2.2 Konstansban megadott tömbméret, dinamikus helyfoglalással

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

#define N 6
#define M 5

typedef int matrix[N][M];

main()
{
   char c;
   matrix *p;
   int i,j;

   clrscr();

   p=(matrix *) calloc(1,sizeof(matrix));
   if (!p)
      {
        printf("Nincs elég memória!");
        return -1;
      }
   for (i=0;i<N;i++)
      {
        for (j=0;j<M;j++)
          {
            (*p)[i][j]=random(10);
            printf("%3d",(*p)[i][j]);
          }
        printf("\n");
      }

   free(p);

   while ((c=getch()) !=13);

}
```

Ebben az esetben futási időben nincs már lehetőségünk a tömb méreteinek változtatására, de ahelyfoglaás itt is csak futási időben történik meg. A hivatkozás majdnem teljesen olyan, mint a mátrixok esetében szokásos. Itt a `calloc` függvényben érdekes módon csak 1 elem számára kell helyet foglalni, ami viszont akkora, mint a tljes mátrix.

2.12.2.3 Mutatótömb

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

#define N 5

main()
{
  char c;

  int *p[N],oszlop[N];
  int i,j;

  clrscr();

  for (i=0;i<N;i++)
    {
      printf("%d. sor oszlopszáma: ",i+1);
      scanf("%d",&oszlop[i]);
    }

  for (i=0;i<N;i++)
    {
      p[i]=(int *) calloc(oszlop[i],sizeof(int));
      if (!p[i])
        {
          printf("Nincs elég memória!");
          return -1;
        }
    }

  for (i=0;i<N;i++)
    {
      for (j=0;j<oszlop[i];j++)
        {
          *(p[i]+j)=random(10);
          printf("%3d",p[i][j]);
        }
      printf("\n");
    }

  for (i=0;i<N;i++)
    free(p[i]);

  while ((c=getch()) !=13);

}
```

Itt a mátrixot egy mutatótömbbel valósítottuk meg. A különlegesség az, hogy a sorok eltérő hosszúságúak is lehetnek. A mutatóknak egyesével adtunk értéket, azaz a helyfoglalás soronként történt. A felszabadítást szintén ciklus segítségével kell végezni.

2.12.3 Struktúrák és mutatók

A korábbiakban már volt szó a strutúrákról. Pointerekkel hivatkozhatunk a struktúrk egyes elemeire is, mint azt a következő példában láthatjuk.

```
struct datum
  { int ev,ho,nap};
struct datum *map;
```

Az így létrehozott mutatónak értéket adhatunk, illetve értékét lekérdezhetjük.

```
(*map).ev=2001; (*map).ho=2; (*map).nap=14;
```

Mivel a C-ben elég gyakori a strutúrák és a pointerek használata, a fenti hivatkozásoknak van egy másik alakja is.

```
map->ev=2001;map->ho=2;map->14;
```

A két hivatkozás egymással teljesen egyenértékű, de a C programokban és a szakirodalomban a második lényegesen elterjedtebb.

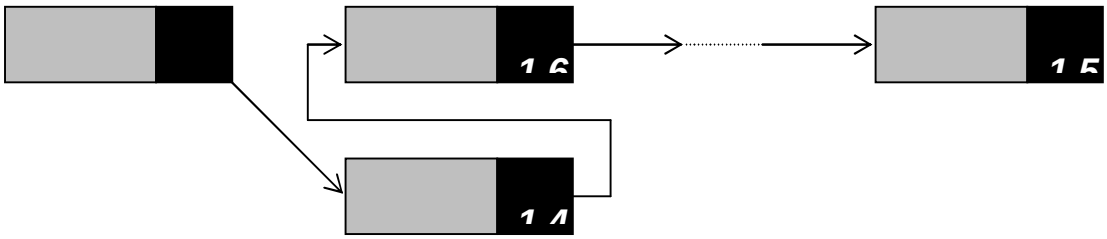
2.12.4 Dinamikus lista

A lista olyan adtaszerkezet, melynek minden egyes eleme két jól elkülöníthető részből áll, az adatrészből és egy mutatóból, mely a lista következő elemére mutat. A lista első elemére a listafejjel hivatkozhatunk. A lista utolsó elemének mutató mezője pedig NULL értékű.

A fenti ábrán a szürke az adatrész a fekete pedig a mutató rész.

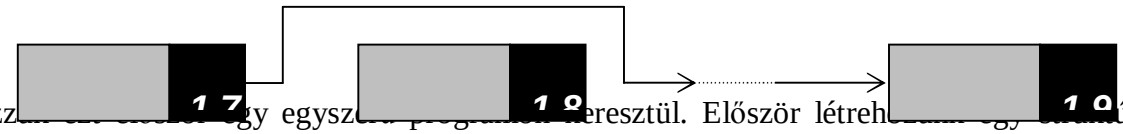
A lista szerkezetnek sok előny van egy vektorhoz képest. Egy listában tnyleg mindig csak annyi elem van, amennyire éppen szükségünk van. A szerkezet módosítása is nagyon egyszerű. Ha egy új adatot akarunk felvenni, akkor

1. megkeressük az új elem helyét
2. az aktuális adat mutaóját átmásoljuk az új elem mutatójába
3. az aktuális elem mutatóját az új elemre irányítjuk



Ha egy listából egy elemet törölni akarunk, akkor a következőképpen járhatunk el:

1. megkeressük a törlendő elemet
2. a mutatóját bemásoljuk az előtte lévő elem mutatójába



Nézzük ezt egy egyszerű programon keresztül. Először létrehozunk egy struktúrát, a struktúra mutató mezője egy ugyanilyen típusú struktúrára hivatkozik, ezt nevezzük *önhivatkozó struktúrának.*

```
#include <stdio.h>
struct lista {
   int value;
   struct lista *next;
};

main()
{
   struct lista n1, n2, n3, n4;
   struct lista *lista_pointer = &n1;

   n1.value = 100;
   n1.next = &n2;
   n2.value = 200;
   n2.next = &n3;
```

```
    n3.value = 300;
    n3.next = &n4;
    n4.value = 400;
    n4.next = 0;


    while( lista_pointer != 0 )  {
       printf("%d\n", lista_pointer->value);
       lista_pointer = lista_pointer->next;
    }
  }
```

Ez még így egyáltalán nem dinamikus, csak a lista használatát figyelhetjük meg rajta. Vegyük észre, hogy az a sok értékadás a listázás előtt, ciklikus folyamat, nyilván nem érdemes ilyen sokszor leírni.

```
  n1.next = n2.next;
  n2_3.next = n2.next;
  n2.next = &n2_3;
```

Mi történik a listával a fenti értékadások hatására?

A következő program már egy dinamikus lista megvalósítására mutat példát.

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

struct data
  {
    int value;
    struct data *nxt;
  };

struct data *head=NULL, *prev, *akt, *next;

void list()
{
  akt=head;
  while (akt!=NULL)
    {
      printf("%5d",akt->value);
      akt=akt->nxt;
    }
}

main()
{
  char c;

  int i=0,sv;

  clrscr();
  randomize();

  printf("Következő szám ");
  scanf("%d",&sv);

  while (sv>0)
    {
      akt=(struct data *) malloc(sizeof(struct data));
      if (!akt)
        {
          printf("Nincs elég memória!");
          return -1;
```

```
        }
      akt->value=sv;
      akt->nxt=NULL;

      if (i==0)
        head=prev=akt;
      else
        {
          prev->nxt=akt;
          prev=akt;
        }
      printf("Következő szám ");
      scanf("%d",&sv);
      i++;
    }

  printf("\n");
  list();

  while ((c=getch()) !=13);
}
```

A főprogramban lévő while ciklussal pozitív számokat olvasunk be, és ezeket fűzzük föl egy dinamikus listába. Látszik, hogy a listában mindig csak a következő elem számára foglalunk helyet a `malloc` függvénnyel. A lista feltöltése után meghívott `list()` függvény a fölvett elemeket listázza ki.

2.13 Makrók

A makrók a függvény kódjába beépített szövegeket jelentenek a C nyelv esetében. Makrók segítségével egyszerű, gyakran alkalmazott műveleteket oldhatunk meg függvények megírása nélkül. C nyelvi makrókat a `#define` direktíva[10] után adhatunk meg. Ezt a direktívát használtuk már korábban kostansok létrehozására.

Az így létrehozott konstansok futásidőben már nem változtathatók[11]. A fordítónak az az első dolga, hogy a forrásnyelvi állományt átadja az előfeldolgozónak. Az előfeldolgozó feladata, hogy a `#define` után talált szövegeket behelyettesítse a forrásprogram azon részébe, ahol hivatkozás történt rájuk. Az előfeldolgozónak lehet, hogy többször is végig kell menni a listán, mivel a makrók minden további nélkül egymásba ágyazhatók

2.13.1 Függvényszerű makrók

A definíció általános alakja:

```
#define azonosító(paraméterek) helyettesítő szöveg
```

A makró hívása:

```
azonosító(argumentumok);
```

Néézünk meg konkrétan néhány függvényszerű makrót!

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

#define min(a,b) ( (a)>(b)?(b):(a))

#define abs(x) ( (x)<0?(-(x)):(x))

#define HA_KICSI(x) (((x)>='a') && ((x)<='z'))
#define NAGY(x) (HA_KICSI(x)?(x)-'a'+'A':(x))
```

[10] A fordítónak szóló utasítás.

[11] A const kulcsszóval definiáltakról ez nem mondható el, ezekkel csak annyit kell tenni, hogy egy rájuk irányítunk egy pointert és ennek az értékét módosítjuk.

```
main()
{
  char c='f';
  int a=10, b=20;
  int k=-3;

  clrscr();

  printf("|%d|=%d\n",k,abs(k));
  printf("%d,%d közül %d a kisebb\n",a,b,min(a,b));
  printf("%c",NAGY(c));

  getch();

}
```

Első látásra föltűnhet, hogy a makródefinícióban látszólag fölösleges helyeken használunk zárójelezést. Mi értelme van annak például, hogy az `abs` makróban a – után az `x`-et zárójelbe tesszük? Induljunk ki abból, hogy az előfeldolgozó csak egyszerű szöveghelyettesítést végez. Azaz, ha a makrót meghívjuk az `a` értékkel, akkor az `x` helyébe `a`-t fog írni. Mi a helyzet, ha a makrót az `a+1` értékkel hívjuk meg? ha zárójelben van az `x`, akkor nincs gond `-(a+1)` kerül behelyettesítésre, ha azonban elhagyjuk a zárójelet, akor a `-a+1` szöveg íródik be, amiről könnyen látható, hogy nem egyezik meg az `a+1` ellentettjével. Tehát függvényszerű makrók írása esetén nagyon fontos a zárójelezés. Inkább legyen fölösleges zárójel, mint hibás működés.

Egy másfajta alkalmazása a függvényszerű makróknak, mikor az általuk átadott érték tokenizálódik[12], vagy pedig az átadott érték szövegként kerül behelyettesítésre. Ezekre az esetekre mutat példát az alábbi program.

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

#define kiir(a) printf("%d\n",(x##a))

#define szoveg(x) printf("\n"#x"\n")

#define szam(x,f) printf("\n"#x"=%"#f"\n",x)

main()
{
  int x1=2,x2=3,x3=4,x4=13;
  int i=13;

  clrscr();

  kiir(1);kiir(2);kiir(3);kiir(4);
  szoveg(Ezt írd ki);
  szam(i,5d);

  getch();
}
```

A kiir() makró bármilyen olyan változónak kiírja az értéket, melynek a neve x-szel kezdődik. A név további részét paraméterként kell átadni.

A szoveg() makró egyszerűen kiírja az átadott szöveget. Figyeljük meg, hogy híváskor nem kell a szöveget idézőjelek közétenni, azt a műveletet a változó neve elé kitett # végzi el.

A szam() makró egy változó nevét és értékét írja ki olyan formátumban, aamilyet a második paramétere előír.

[12] Két szintaktikai egységből egyet készítünk. Egy változónévhez hozzávszünk még néhány karaktert, ezzel a változó nevét módosítjuk.