

1 Tartalomjegyzék

1	TARTALOMJEGYZÉK	1
2	C PROGRAMOZÁS KÖZÉPISKOLÁSOKNAK	3
2.1	AZ ELSŐ C PROGRAM	3
2.2	MEGJEGYZÉSEK.....	3
2.3	A VÁLTOZÓKRÓL.....	3
2.3.1	<i>Felhasználói típus</i>	5
2.4	KONSTANSOK.....	6
2.4.1	<i>A const kulcsszó használatával</i>	6
2.4.2	<i>Az előfordítónak szülő helyettesítéssel</i>	6
2.5	OPERÁTOROK	6
2.5.1	<i>Aritmetikai operátorok</i>	7
2.5.2	<i>Összehasonlító és logikai operátorok</i>	7
2.5.3	<i>Léptető operátorok</i>	7
2.5.4	<i>Bitműveletek</i>	7
2.5.5	<i>Értékadó operátorok</i>	8
2.5.6	<i>Feltételes operátor</i>	8
2.5.7	<i>Pointer operátorok</i>	8
2.5.8	<i>Típuskonverziók</i>	8
2.6	ADATOK BEOLVASÁSA A BILLENTYŰZETRŐL	9
2.7	ITERÁCIÓK.....	10
2.7.1	<i>while ciklus</i>	10
2.7.2	<i>for ciklus</i>	11
2.7.3	<i>do-while ciklus</i>	12
2.8	SZELEKCIÓK	12
2.8.1	<i>if utasítás</i>	12
2.8.2	<i>if-else szerkezet</i>	13
2.8.3	<i>switch utasítás</i>	13
2.9	TÖMBÖK.....	14
2.9.1	<i>Egydimenziós tömbök</i>	14
2.9.2	<i>Sztringek</i>	15
2.9.3	<i>Kettő és több dimenziós tömbök</i>	17
2.10	FELHASZNÁLÓ ÁLTAL DEFINIÁLT TÍPUSOK	18
2.10.1	<i>Struktúrák</i>	18
2.10.2	<i>Struktúrát tartalmazó tömbök</i>	19
2.11	FÜGGVÉNYEK.....	19
2.11.1	<i>Paraméterátadás</i>	19
2.11.2	<i>Lokális és globális változók</i>	20
2.11.3	<i>Automatikus és statikus változók</i>	20
2.11.4	<i>Register módosító jelző</i>	21
2.11.5	<i>Vektor átadása függvénynek</i>	21
2.12	MUTATÓK (POINTEREK).....	22
2.12.1	<i>Vektorok és mutatók</i>	23
2.12.2	<i>Kétdimenziós tömbök és pointerek</i>	24
2.12.2.1	<i>Dinamikus tömb egy vektorban</i>	24
2.12.2.2	<i>Konstansban megadott tömbméret, dinamikus helyfoglalással</i>	25
2.12.2.3	<i>Mutatótömb</i>	26
2.12.3	<i>Struktúrák és mutatók</i>	27
2.12.4	<i>Dinamikus lista</i>	27
2.13	FILE KEZELÉS	29

2.13.1	<i>File előkészítése, lezárása</i>	30
2.13.2	<i>Szöveges állományok</i>	31
2.13.3	<i>Bináris állományok</i>	32
2.14	A SZÖVEGES KÉPERNYŐ KEZELÉSE	33
2.14.1	<i>Szöveges üzemmódban gyakran használt függvények:</i>	34
2.14.2	<i>Képernyő mentése és vissztöltése</i>	35
2.14.3	<i>Néhány karaktert tesztelő makró</i>	36
2.14.4	<i>Karaktert átalakító függvények</i>	37
2.14.5	<i>Néhány konverziós függvény</i>	37
2.15	A GRAFIKA PROGRAMOZÁSA	37
2.15.1	<i>Koordináta-rendszer</i>	37
2.15.2	<i>Grafika indítása</i>	38
2.15.3	<i>Gyakran használt függvények</i>	39
2.16	MAKRÓK	40
2.16.1	<i>Függvényszerű makrók</i>	40
3	FELADATGYŰJTEMÉNY	42
4	FÜGGELÉK	51

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.

1., 2. feladat

2.2 Megjegyzések

A megjegyzéseket a programszöveg magyarázatainak 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 változokról

A C típusos programnyelv. Ez azt jelenti, hogy mielőtt egy változót használni szeretnénk deklarálnunk kell azt. Figyeljünk arra, hogy 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 értékészlet módosul.

Pl.

```
int a  
unsigned1 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 signed² módosító jelző is használható, de alapértelmezés szerint minden egész változó ilyen lesz.

¹ Előjel nélküli

² Előjeles

A következő táblázatban felsoroltunk néhány típust, értékészletükkel 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+308	8	15
long double	3.4e-4932 .. 3.4e+4932	10	19

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

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

Lehetőség van arra is, hogy meghatározzuk a változó értéke által elfoglalt mező szélességét. Nézzük 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 ebben 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 szeretnénk volna adni.

2.3.1 Felhasználói típus

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

typedef típus típusnév

Egy konkrét példán keresztül

```
typedef unsigned long int eszesz;  
eszesz n;
```

Néhány további formázási lehetőség a printf()-ben:

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

3., 4., 5., 6. feladat

2.4 Konstansok

A konstansok megadásának két módját ismertetjük az alábbiakban.

2.4.1 A const kulcsszó használatával

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

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

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

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

2.4.2 Az előfordítónak szóló helyettesítéssel

Az előfordítónak különböző definíciókat, leírásokat adhatunk, erről a későbbiekben 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 az #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 operandusok, 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 alpművelet, 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 kifejezésekben 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 komplementum, & 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 `a` értéke 4, `b` értéke pedig 13 lesz. A kiértékelés 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
<code>a=a+b</code>	<code>a+=b</code>
<code>a=a-b</code>	<code>a-=b</code>
<code>a=a*b</code>	<code>a*=b</code>
<code>a=a/b</code>	<code>a/=b</code>
<code>a=a%b</code>	<code>a%=b</code>
<code>a=a<<b</code>	<code>a<<=b</code>
<code>a=a>>b</code>	<code>a>>=b</code>
<code>a=a&b</code>	<code>a&=b</code>
<code>a=a b</code>	<code>a =b</code>
<code>a=a^b</code>	<code>a^=b</code>

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 megemlíttünk 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 információvesztés 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 műveletek 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

7., 8., 9. feladat

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 megfelelőek, 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 néhány 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 közöttük 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

10. feladat

2.7 Iterációk

Mint minden magas szintű programozási nyelvben a C-ben is vannak olyan utasítások, melyek egy feltételtől függően többször is végrehajtják 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 utasításokat 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 kifejezés 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);
}
```

11., 12., 13. feladat

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 címkék 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 elválasztva 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.

14., 15., 16., 17., 18., 20. feladat

2.9 Tömbök

Az eddigiek során olyan változókat használtunk csak, melyek egy érték tárolására voltak alkalmasak. Gyakran van azonban szükségünk arra, hogy ugyanazzal a változónévvel több értékre is tudjunk hivatkozni. Ebben az esetben használjuk a tömböket. (Egy osztály minden tanulóához 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.

A tömb méretét akár futásidőben is meg tudjuk mondani. Ha egy program tesztelése során változtatjuk egy tömb méretét (új elemeket veszünk fel a fősorolásba), 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++)
    {
        printf("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' };

```

Értékkadás 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 sztring kezelő 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 sztringbe másolja az sz2 sztringet	sz2="world" strcat(sz1,sz2); printf(" %s\n",sz1)

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

		Eredménye: world
strcmp(sz1,sz2)	Összehasonlítja a két sztringet, ha egyenlők, akkor , ha nem egyenlők, akkor nem 0 értékkel tér vissza, különben igen.	sz1="Hello"; sz2="HeLl0" n=strcmp(sz1,sz2); printf("%d\n",n)
		Eredménye: 32
strncmpi(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 sztringben először hol fordul elő a c karakter	char c='e', sz1[]="Hello", *ch; ch=strchr(sz1,c); printf("%d\n", ch-sz1);
		Eredménye: 1⁴
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 sztringhez 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 sztring kezelő függvényeket használni akarjuk,akkor mindenképpen szükség van a `#include <string.h>` **beszúrá**sra.

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

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

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!


```

main()
{
    int sz[]="C programozási nyelv";
    int i=0;
    while (s[i])
    {
        printf("\n%c",s[i];
        i++;
    }
}

```

21., 22., 23., 24., 25., 26. feladat

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 példaként egy valós értékeket tartalmazó két dimenziós tömbre:

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

Az első index ebben az esetben 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");
}
}
```

27. feladat

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. Ezeket leggyakrabban adatbázisokban találjuk meg. Gondoljunk itt egy személy kereszt és vezetéknévére, 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 példában leírt struktúra:

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

Ha ilyen típusú változót akarunk létrehozni, akkor annak módja:

```
struct személy sz;
```

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

```
sz.vnev="Kovács";
```

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

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

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

```
személy 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;
```

28. feladat

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 strukturáltan, jól olvashatóan szeretnénk megírni. Ez nagyon fontos a későbbi módosíthatóság miatt.
- Természetesen az első kettő eset együtt is fennállhat.

A függvény definíció általános alakja:

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

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 érték szerint. 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 operátort 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.

29., 30., 31. feladat

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;
}

```

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 a Pascal lokális változói között.

```

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

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

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

```

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 statikus 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 automatikus 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 adódhat 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 egészet 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 indirektség 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 p egy egész típusú változóra mutató pointer */
p=&a; /* p az a címét veszi föl értéként */
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 vektorok és az egyszeres indirektségű⁷ 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];
```

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 kellett 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 tehattünk 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, kírja 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)
```

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 a mutató 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.

```

    {
        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` függvélynél meg kell mondanunk, hogy hány elem számára szeretnénk helyet foglalni, és hogy egy elemnek mekkora a mérete, ebben a sorrendben. A függvény a lefoglalt memóriaterületet rögtön feltölti nullával. A `malloc` függvénnel pedig azt kell közölni, hogy mekkora memóriaterületet szeretnénk lefoglalni (byte-ban).

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

Figyeljük meg a `p`-nek történő értékadást! A `calloc` függvény visszatérési értéke egy típus nélküli mutató, ezt egy típuskonverzióval át kellett alakítani a `p` típusának megfelelő alakra.⁹ Minden esetben meg kell vizsgálni, 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;

```

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


```

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++)
{
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 memóriefoglalás ebben a programban valóban futási időben történik. Ha közelebbről is szemügyre vesszük a listát látható, hogy itt a mátrixot valójában egy vektorban tároljuk. Az aktuális pozíciót pedig a

$hely = \text{aktuális_sor} * \text{oszlopszám} + \text{aktuális_oszlop}$

formával határoztuk meg. Figyeljünk fel 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 a helyfoglalás 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 teljes 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.

33., 34. feladat

2.12.3 Struktúrák és mutatók

A korábbiakban már volt szó a struktúrákról. Pointerekkel hivatkozhatunk a struktúrák 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 lekérdezhethetjük.

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

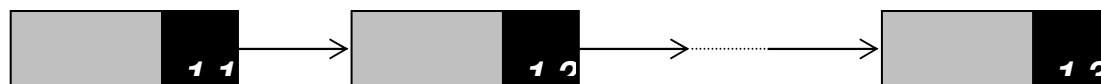
Mivel a C-ben elég gyakori a struktúrák és a pointerek használata, a fenti hivatkozásoknak van egy másik alakja is.

```
map->ev=2001;map->ho=2;map->nap=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 adtászerkezet, 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 listafejvel hivatkozhatunk. A lista utolsó elemének mutató mezője pedig NULL értékű.



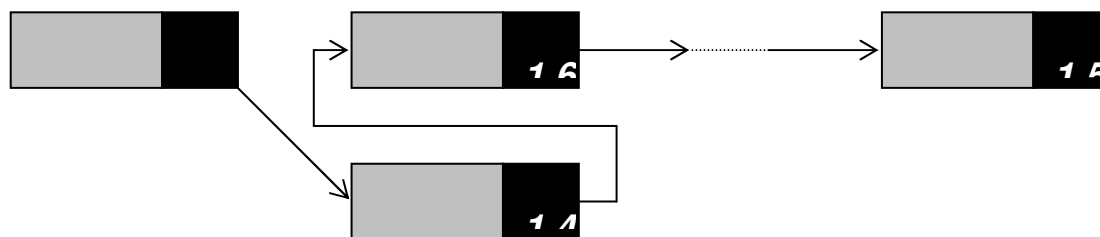
A fenti ábrán a szürke az adatrész a fekete pedig a mutató rész.

A lista szerkezetnek sok előny van egy vektorhoz képest. Egy listában tényleg 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

az aktuális adat mutatóját átmásoljuk az új elem mutatójába

2. az aktuális elem mutatóját az új elemre irányítjuk



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

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

Nézz meg egy egyszerű listát a következő ábrán keresztül. Először létrehozunk egy listát, 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 fölvettem elemeket listázza ki.

35. feladat

2.13 File kezelés

A C nyelvben a data állományokat tartalmuk alapján két csoportra oszthatjuk, szöveges és bináris állományokra. A szöveges állományok olvasható információt tartalmaznak. Sorokból épülnek fel, minden sor végét a CR/LF karakterpár zárja. A bináris állományok byte-okból felépülő adtahalmazt jelentenek. A szöveges állomány minden esetben földolgozható, mint bináris állomány, de ez fordítva már nem igaz.

Szöveges állományokban is lehet számokat tárolni. Ebben az esetben a számok, mint karakterek fogan tárolódnak. Ha például egy négy jegyű számot szöveges állományban tárolunk, akkor négy byte-nyi helyet foglal el, ha azonban egy bináris állományba tesszük le, akkor csak 2 byte kell. (Természetesen csak akkor, ha int típusú volt)

2.13.1 File előkészítése, lezárása

A file kezeléshez tartozó függvények az `stdio.h` deklarációs állományban vannak leírva. Minden file-hoz hozzá kell rendelni egy FILE típusú mutatót, mely a memóriában a file jellemzőire mutat¹⁰. Ennek formája:

```
FILE *fp;
```

Ez után következhet a file megnyitása:

```
fp=fopen("A:\SZOVEG\NEV.TXT"név", "mód");
```

Az `fp` mutató értéke `NULL`, ha az állomány megnyitása sikertelen volt. A fizikai file névben teljes elérési útvonalat is megadhatunk, ebben az esetben azonban ügyelni kell a `\` jel használatára:

```
"A:\SZOVEG\NEV.TXT" helyett "A:\\SZOVEG\\NEV.TXT"11
```

A mód paraméter azt jelenti, hogy milyen műveleteket akarunk végezni az állománnyal.

Mód	Leírás
r (+)	Létező file megnyitása olvasásra. File mutató a file elejére áll.
w (+)	Új file megnyitása írásra. Létező file esetén annak tartalma elvész. File mutató a file elejére áll.
a (+)	File megnyitása hozzáírásra. Nyitás után a file mutató a file végére áll. Ha a file nem létezik, akkor az <code>fopen</code> létrehozza

Mindegyik mód jellemző kiegészíthető egy `+` paraméterrel, ez minden esetben azt jelenti, hogy az állományt olvashatjuk és írhatjuk is. A mód sztringben szoktuk még megadni, hogy milyen állománnyal dolgozunk. `t` szöveges, `b` bináris állományt jelent.

Nézzünk egy konkrét példát, melyben egy szöveges állományt megnyitunk írásra és olvasásra:

```
FILE *fp;
fp=fopen("A:\\SZOVEG\\NEV.TXT", "r+t");
if (fp==NULL)
{
    printf("Sikertelen file nyitás!");
    return -1;
}
```

A file nyitást gyakran szokták az `if`-en belülre helyezni. Ekkor a szintaktikája a következő:

```
if (!(fp=fopen("A:\\SZOVEG\\NEV.TXT", "r+t")))
{
    printf("Sikertelen file nyitás!");
    return -1;
}
```

Ha a file-on elvégeztük a megfelelő műveleteket akkor le kell zárni. Ezt az `fclose(fp)` függvényhívás valósítja meg. Amennyiben egy fájlban írási műveleteket is végeztünk lezárás előtt célszerű az `fflush(fp)` függvényhívás, ez az átmeneti pufferek tartalmát kiírja az állományba, ha még nem történt volna meg.

¹⁰ Logikai file név

¹¹ A string konstansokban a `\` jel egy formázó karaktert vezet be.

2.13.2 Szöveges állományok

Szöveges állományokkal kapcsolatban leggyakrabban alkalmazott függvényeket az alábbi táblázatban foglaljuk össze. Minden esetben a

```
FILE *fp;
char ch, string[]="Kiírandó szöveg";
char sz[20];
definíciókat használjuk
```

Függvény	Leírás
ch=fgetc(fp)	Beolvas egy karaktert a file-ból
fputc(ch, fp)	Kiír egy karaktert a file-ba
fgets(sz, strlen(string)+1, fp)	Az sz változóba beolvas egy sztringet
fputs(string, fp)	A string változó tartalmát kiírja a file-ba
fscanf(fp, "konv", &valt)	Hasonló a scanf-hez, eltérés az első paraméterben
fprintf(fp, "konv", valt)	Hasonló a printf-hez, eltérés az első paraméterben

Az alábbiakban két egyszerű programot mutatunk be a szöveges állományok kezelésére:

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

main()
{
    char c;
    FILE *fp;
    int i;

    clrscr();

    if (!(fp=fopen("A:\\Fileok\\p.txt", "wt")))
    {
        fprintf(stderr, "Nem sikerült megnyitni az állományt");
        return -1;
    }

    for (i=0; i<10; i++)
        fprintf(fp, "%4d", i);
    fflush(fp);
    fclose(fp);

    if (!(fp=fopen("A:\\Fileok\\p.txt", "rt")))
    {
        fprintf(stderr, "Nem sikerült megnyitni az állományt");
        return -1;
    }

    i=0;
    while (!felt(fp))
    {
        fscanf(fp, "%d", &i);
        printf("%d", i);
    }
    fclose(fp);
    while ((c=getch()) !=13);
}
```

A példában egész értékeket írunk ki egy szöveges állományba, majd visszaolvassuk azokat. A visszaolvasásánál a file vége jelölésére az `fgetc(fp)` függvényt használjuk. Ez a file végére érve vesz föl `NULL` értéket.

A másik példa egy állomány nevét kéri be, majd karakterenként kilistázza a képernyőre.

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

main()
{
    char nev[25];
    FILE *fp;
    char c;

    clrscr();

    printf("Állomány neve: ");
    scanf("%24s", nev);
    clrscr();

    fp = fopen ( nev, "rt");

    if( fp == NULL )
        printf("Nem lehet megnyitni a(z) %s állományt\n", nev);
    else {
        while( (c = fgetc(fp)) != EOF )
            putchar (c);
        fclose (fp);
    }
}
```

2.13.3 Bináris állományok

A bináris állományokat byte-onként vagy blokkonként kezelhetjük. A byte-onkénti kezeléshez jól használható az előző részben leírt `fgetc` és `fputc` függvénypáros. A blokkonkénti kezelést pedig az `fread` és az `fwrite` függvényekkel végezhetjük el.

Használatukat a következő példában figyelhetjük meg.

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

main()
{
    char c;
    FILE *bf;
    int i[10]={1,2,3,4,5,6,7,8,9,10}, counter, sv;

    clrscr();

    if (!(bf=fopen("A:\\fileok\\adat.dat", "w+b")))
    {
        printf("Sikertelen file nyitás");
        return -1;
    }

    fwrite(i, sizeof(int), 10, bf);

    rewind(bf);
    counter=fread(&sv, sizeof(int), 1, bf);
    printf("%4d", sv);
}
```



```

fseek (bf, 3*sizeof(int), SEEK_SET); /* A 4. elemre áll */
counter=fread(&sv, sizeof(int), 1, bf);
printf ("%4d", sv);

fseek (bf, 0*sizeof(int), SEEK_CUR); /* A rákövetkezőre áll */
counter=fread(&sv, sizeof(int), 1, bf);
printf ("%4d", sv);

fclose (bf);

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

```

Nézzük először az `fwrite` függvényt. A függvény első paramétere a kiírandó változó neve, második paramétere az adott struktúrához tartalmazó blokkméret, harmadik paraméter a kiírandó elemek darabszáma, negyedik pedig az állomány mutatója.

Az `fread` függvény hasonlóképpen paraméterezhető. Eltérés abban van hogy beolvasáskor nyilván a változó címét kell megadni a függvénynek, a másik három paraméter ugyanaz. Természetesen mindkét függvénynek van visszatérési értéke is. Mindkettőnél a sikeresen kiírt illetve beolvasott blokkok számát adja vissza. Ez az érték használható a sikertelen beolvasás illetve kiírás figyelésére. Ha a fenti programban a `counter` változó értéke nem egyenlő eggyel, akkor az azt jelenti, hogy nem sikerült beolvasni adatot az állományból. Ezeknek a változóknak a megadása nem kötelező, mint a fenti programból is látszik az `fwrite` függvényt anélkül hívtuk meg.

A programban van még egy újdonság, az állományban való pozícionálás. Erre az `fseek` függvényt használjuk. Paraméterezése: első paraméter a file mutató, második a blokkméret és egy egész szám szorzata (ennyiedik elemre fogunk állni), harmadik pedig azt mutatja meg, hogy mihez viszonyítjuk a pozícionálást. `SEEK_CUR` esetén a pillanatnyi pozícióhoz, `SEEK_SET` esetén pedig a file elejéhez. Vigyázzunk azonban ennek az alkalmazásával, a sorszámozás most is a nulladik elemtől kezdődik, ennek a `SEEK_SET` esetén lehet jelentősége. A pillanatnyi pozícióval pedig az a helyzet, hogy amikor egy adat beolvasása megtörtént, akkor a file mutató rögtön eggyel tovább lép, tehát, ha a szomszédos elemre akarunk lépni, akkor természetesen a `fseek`-ben 0-t kell adnunk a második paraméter helyén¹².

36., 37., 38. feladat

2.14A szöveges képernyő kezelése

A Turbo C a képernyőt grafikus és szöveges üzemmódban működtethető. Mindkét üzemmódban nagyon sok függvényt segít változatosabbá tenni a képernyőt ebben a fejezetben a szöveges üzemmód függvényeiről lesz szó.

Ha ezeket a függvényeket használni szeretnénk, akkor a program elején a következő header file-okra lesz szükségünk:

```

#include <conio.h>
#include <string.h>
#include <ctype.h>

```

Valójában a képernyőt kezelő függvények az elsőben vannak deklarálva, de képernyőn sztringekkel és karakterekkel dolgozunk, ezért szinte mindig szükségünk van a másik kettőre is. A sztring kezelő függvényekről korábban már volt szó ezt még ebben a fejezetben kiegészítjük néhány konverziós függvénnyel és a gyakrabban használt karakter kezelő függvényeket is felsoroljuk.

¹² Ilyet nyilván nem fogunk csinálni, mert automatikus a továbblépés

2.14.1 Szöveges üzemmódban gyakran használt függvények:

Név	Leírás	Név	Leírás
cprintf()	Formázott kivitel, hasonló a printf-hez	clreol()	sor törlése az aktuális kurzor pozíciótól
cputs()	egy string kivitele a képernyőre	gotoxy()	a képernyő adott pontjába mozgatja a kurzort
putch()	egy karakter kivitele	window()	ablak definiálása
getche()	egy karakter beolvasása a billentyűzetről és kiírása a képernyőre	textcolor()	szöveg színe
clrscr()	képernyő törlése	textbackground	háttér színe

Ha a képernyőn egy ablakot hozunk létre, akkor onnantól kezdve minden koordinátát az ablakhoz képesti relatív értékekkel kell megadnunk. Ha azt akarjuk, hogy az ablak színe más legyen, mint a környezeté, akkor a háttérszín beállítása után törölnünk kell a képernyőt.

Az aktív ablakra vonatkozó információkat is lekérdezhethetjük. Ehhez a rendszer egy a `conio.h`-ban definiált struktúrát bocsát rendelkezésünkre

```
struct text_info {
    unsigned char winleft;          /* az ablak bal felső sarka x koord */
    unsigned char wintop;          /* az ablak bal felső sarka y koord */
    unsigned char winright;        /* az ablak jobb alsó sarka x koord */
    unsigned char winbottom;       /* az abla jobb alsó sarka y koord */
    unsigned char attribute;       /* a szöveg attribútuma szín+háttér együtt */
    unsigned char normattr;        /* normál attribútum */
    unsigned char currmode;        /* aktuális mód: */
    unsigned char screenheight;    /* a képernyő magassága */
    unsigned char screenwidth;     /* a képernyő szélessége */
    unsigned char curx;            /* a kurzor aktuális pozíciója x koord */
    unsigned char cury;            /* a kurzor aktuális pozíciója y koord */
};
```

Attól, hogy a struktúrában karakterekként vannak létrehozva az egyes tagok nyugodtan végezhetünk velük matematikai műveleteket, de az is lehetséges, hogy egy egész típusú változóhoz rendeljük hozzá a lekérdezett értékeket. Ha ezt a struktúrát használatba szeretnénk venni, akkor szükség van a

```
struct text_info info;
gettextinfo(&info);
```

definíciókra is. Ezután már minden további nélkül használható az

```
x=info.screenwidth;
```

típusú hivatkozás.

Nézzünk egy példa programot a képernyő kezelésére. Az alábbi program az aktuális szöveges képernyő közepére rajzol egy 40x20-as ablakot. Ebbe 5 menüpontot ír ki, és közülük beolvas egyet. A program futásának csak akkor van vége, ha tényleg létező pontot olvastunk be.

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

#define SZ 20
#define M 10
```

```

char menu_pontok[5][15]={"Első pont", "Második pont", "Harmadik pont", "Negyedik
                        pont", "V É G E"};
char cim[]=" M E N Ü P O N T O K ";
char ala[]="-----";
char val[]="Kérem válasszon: ";

main()
{
    char c;
    int xj,yj,xb,yb;
    int i;
    char valasz;

    struct text_info info;
    gettextinfo(&info);
    textbackground(BLACK);
    clrscr();

    xb=info.screenwidth/2-SZ;
    yb=info.screenheight/2-M;
    xj=xb+2*SZ;
    yj=yb+2*M;

    window(xb,yb,xj,yj);
    textbackground(YELLOW);
    textcolor(BLUE);
    clrscr();
    gotoxy(20-strlen(cim)/2,2);
    cprintf(cim);
    gotoxy(20-strlen(cim)/2,3);
    cputs(ala);

    for (i=0;i<5;i++)
    {
        gotoxy(12,2*i+6);
        cprintf("%2d.  %s",i+1,menu_pontok[i]);
    }

    do
    {
        gotoxy(12,2*M-3);
        cprintf(val);
        valasz=getche();
        clreol();
    }
    while(!isdigit(valasz) || valasz>'5' || valasz<'1' );

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

```

Tanulmányozzuk át figyelmesen a programot! Vegyük észre, hogy szöveg kiírására a `cputs` ugyanolyan alkalmas, mint a `cprintf`! A `valasz` beolvasásakor találunk egy olyan függvényt, mely az eddigiekben még nem szerepelt, `isdigit(valasz)`, visszatérési értéke nem nulla, ha a `valasz` értéke számjegy.

2.14.2 Képernyő mentése és visszatöltése

Egy jól használható lehetősége a szöveges képernyőkezelő függvényeknek az ablak teljes egészének, vagy egy részletének elmentése, majd visszatöltése a memóriába. A következő program erre mutat egy példát:

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

```

char puffer[4096];

int main(void)
{
    int i;

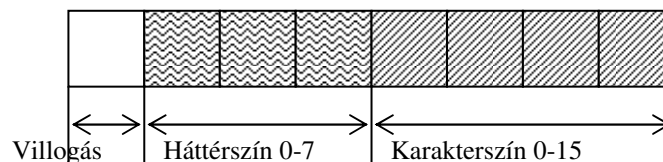
    clrscr();
    for (i = 0; i <= 20; i++)
        cprintf("Sorszama %d\r\n", i);
    gettext(1, 1, 80, 25, puffer);

    gotoxy(1, 25);
    cprintf("Képernyőtörlés gombnyomásra...");
    getch();
    clrscr();
    gotoxy(1, 25);
    cprintf("Az előző tartalom visszatöltése...");
    getch();
    puttext(1, 1, 80, 25, puffer);
    gotoxy(1, 25);
    cprintf("Program vége...");
    getch();
}

```

Ehhez a `gettext` és a `puttext` függvényeket használjuk. Egy billentyű lenyomására várakozik a `getch()` függvény. Figyeljük meg a `puffer` méretét! Ha a $80 \times 25 = 2000$ byte-ból indulunk ki, akkor úgy tűnik, mintha kétszer akkora területet foglaltunk volna le, ez azonban nincs így, mert nemcsak azt kell tudnunk, hogy mi jelenik meg a képernyő egy adott pontján, hanem azt is, hogy milyen színű háttéren és milyen karakterszínnel. Ezért minden egyes hely tárolásához 2 byte-ra van szükség. 1 byte a tartalom, 1 byte az attribútum.

Az attribútum byte fölépítése:



2.14.3 Néhány karaktert tesztelő makró

Minden makró¹³ argumentumában `char` típusú változó szerepel és mindegyik visszatérési értéke `int`.

Makró	Leírás	Makró	Leírás
<code>isalnum</code>	☒ betű ('A'-'Z', 'a'-'z') vagy számjegy ('0'-'9')	<code>isxdigit</code>	☒ hexadecimális számjegy ('0'-'9', 'A'-'F', 'a'-'f')
<code>isalpha</code>	☒ betű ('A'-'Z', 'a'-'z')	<code>isprint</code>	☒ nyomtatható karakter
<code>isascii</code>	☒ alsó byte-ja 0 és 127 közé esik	<code>islower</code>	☒ kisbetű
<code>iscntrl</code>	☒ vezérlőkarakter	<code>isspace</code>	☒ szóköz, CR, LF, tab (vissz., függ.) lapdobás
<code>isdigit</code>	☒ számjegy ('0'-'9')	<code>isupper</code>	☒ nagybetű
<code>isgraph</code>	☒ nyomtatható karakter, de nem szóköz	<code>ispunct</code>	☒ elválasztó karakter, az összes nyomtatható karakter a betűk, számok és szóköz nélkül.

☒ jelentése: a makró visszatérési értéke nem nulla, ha c értéke

¹³ A meghívásból nem látszik, hogy makróról vagy pedig függvényről van szó. A későbbiek során még szót ejtünk a makrókról is. Egyelőre elég annyi, hogy ezek nem függvények, hanem makrók.

2.14.4 Karaktert átalakító függvények

Függvény	Leírás
toascii	ASCII karakterré alakít
toupper	Az angol kis betűket nagy betűkké alakítja
tolower	Az angol nagybetűket kis betűkké alakítja

2.14.5 Néhány konverziós függvény

Függvény példával	Leírás
<pre>float f; char *str = "12345.67"; f = atof(str);</pre>	Sztringet konvertál át lebegőpontos értéké, sikertelen esetén 0-val tér vissza
<pre>int n; char *str = "12345.67"; n = atoi(str);</pre>	Sztringet konvertál át egész számmá, sikertelen esetén 0-val tér vissza
<pre>long l; char *lstr = "98765432"; l = atol(lstr);</pre>	Sztringet konvertál át hosszú egész számmá, sikertelen esetén 0-val tér vissza
<pre>char *string = "87654321", *endptr; unsigned long lnumber; lnumber = strtoul(string, &endptr, 10);</pre>	Sztringet egészzé alakít, az utolsó szám a számrendszer alapszáma (2-36), endptr a hibára mutat
<pre>char *string = "87654321", *endptr; long lnumber; lnumber = strtol(string, &endptr, 10);</pre>	Az előzőhöz hasonló, csak a visszatérési érték típusa más
<pre>int number = 12345; char string[25]; itoa(number, string, 10);</pre>	Egész számot sztringgé konvertál, utolsó jegy az alapszám
<pre>char string[25]; long value = 123456789L; ltoa(value, string, 10);</pre>	Hosszú egészet alakít sztringgé
<pre>unsigned long lnumber = 3123456789L; char string[25]; ultoa(lnumber, string, 10);</pre>	Előjel nélküli hosszú egészet alakít sztringgé

39., 40. feladat

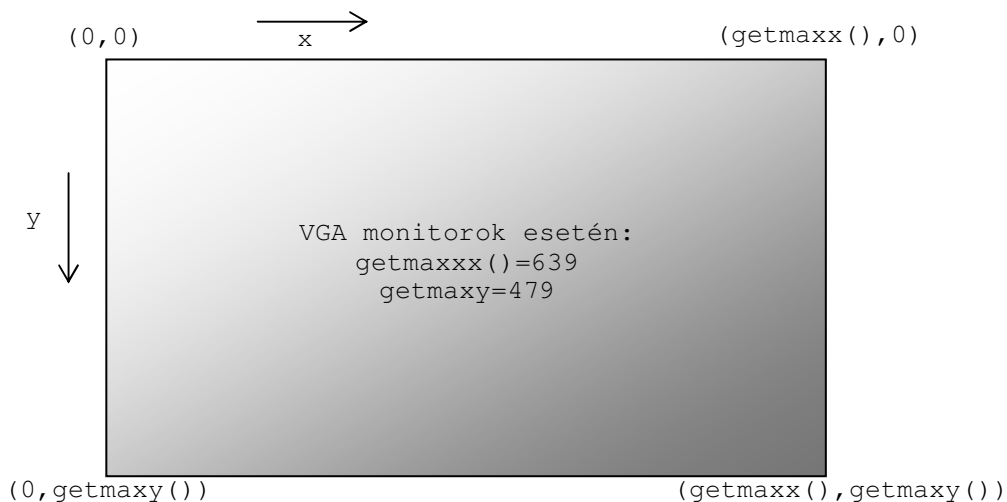
2.15A grafika programozása

A grafikus függvénycsomagban jól használható függvények csoportja áll rendelkezésünkre ahhoz, hogy egyszerűbb pixel grafikus ábrákat készítsünk. Valójában minden alakzat, amit a képernyőn megjelenítünk a `putpixel` függvényre támaszkodik, de a rajzolás megkönnyítése érdekében sok grafikus primitív¹⁴ áll rendelkezésünkre. (egyenes, kör téglalap, ellipszis, ív stb rajzolása) Ha a grafikus függvényeket használni akarjuk, akkor be kell emelni `graphics.h` header file-t. Grafika programozásánál gyakran használjuk a matematikai függvényeket is, ezért általában az elején hozzátesszük a programhoz a `math.h` állományt is.

2.15.1 Koordináta-rendszer

Mielőtt hozzáfognánk a grafika programozásához meg kell ismernünk a rendszer által alkalmazott grafikus koordináta-rendszert.

¹⁴ Olyan függvény, mely egy megfelelő paraméterezéssel az adott alakzatot megrajzolja. Ha egy rendszerben sok grafikus primitív van, akkor az megkönnyíti a grafika programozását. Grafikus primitív például egy kör rajzolása, de nem az olimpiai öt karika megrajzolása. Persze, ha mi magunk írunk egy ilyen függvényt, akkor arra a továbbiakban, mint egy grafikus primitívre lehet hivatkozni.



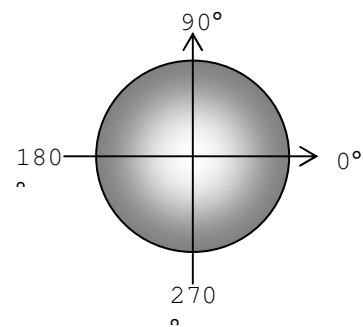
A matematikában alkalmazott koordináta-rendszer ettől egy kicsit eltér. Számunkra az lenne a természetes, ha az origó a képernyő középpontjában lenne, és az y értékek alulról fölfelé növekednek. Ha ezt a koordináta-rendszert szeretnénk használni, akkor szükség van egy koordináta-transzformációra.

```
void trafoxy(int *x,int *y)
{
    *x+=getmaxx() / 2;
    *y=getmaxy() / 2 - *y;
}
```

Tehát ha a matematikai koordinátákkal akarunk dolgozni, akkor számolásakor azokat használjuk, majd ábrázolásakor a fenti függvénnyel áttranszformáljuk azokat képernyő-koordinátákká. A függvény hívásának módja:

```
trafoxy(&x,&y);
```

Ha egy grafikus függvény szögekkel dolgozik, akkor azt fokokban kell megadni, a matematikában megszokott körüljárás szerint.



2.15.2 Grafika indítása

Mielőtt grafikai függvényeket használnánk elő kell készítenünk a képernyőt. Ennek módját mutatjuk meg az alábbiakban.

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

main()
{
    int Gd, Gm, Hiba;

    Gd=DETECT;
    initgraph(&Gd, &Gm, "c:\\progra~1\\bc\\bgi");
    Hiba=graphresult();
    if (Hiba)
    {
        cprintf("Grafikus hiba: %s", grapherrormsg(Hiba));
        exit(1);
    }

    rectangle(100,100,60,40);
    getch();
}
```

```
closegraph();
```

```
}
```

DETECT segítségével földerítjük az aktuális monitor jellemzőit, majd inicializáljuk a grafikát, az idézőjelek között a BGI file-ok helyét¹⁵ kell megadnunk a programnak, ezen állományok segítik a képernyő grafikus üzemmódban való kezelését. A `graphresult()` függvény jelzi, ha valami miatt nem sikerült az áttérés grafikus módra. Ha minden rendben volt következhetnek a grafikai függvényhívások, (itt ezt a `rectangle()` függvény jelzi), ha végeztünk, akkor pedig a `closegraph();` függvénnyel lezárjuk a grafikát.

2.15.3 Gyakran használt függvények

Az alábbiakban egy táblázatban soroljuk föl a leggyakrabban használt grafikus függvényeket, a teljesség igénye nélkül. (A rendszerben közel 80 grafikus függvény van)

Függvény	Leírás
<code>setcolor(c)</code>	A rajzolás színét állítja be c 0-15 közötti érték, vagy színkonstans
<code>setbkcolor(c)</code>	A háttér színét állítja be, c ua., mint előbb
<code>setlinestyle(ls,pt,th)</code>	ls értéke a vonalstílusára utal: SOLID_LINE, DOTTED_LINE, CENTER_LINE, DASHED_LINE, USERBIT_LINE ebben az esetben kerül értelmezésre a pt ez egy 16 bites minta, ezt ismétli a vonalon th a vonalvastagság, NORM_WIDTH, THICK_WIDTH
<code>setfillstyle(pt,c)</code>	A kitöltési mintát adja meg pt-ben, c-ben pedig a színt. EMPTY_FILL, SOLID_FILL, LINE_FILL, LTSLASH_FILL, SLASH_FILL, BKSLASH_FILL, LTBBKSLASH_FILL?, HATCH_FILL, XHATCH_FILL, INTERLEAVE_FILL, WIDE_DOT_FILL, CLOSE_DOT_FILL, USER_FILL
<code>moverel(dx,dy)</code>	A grafikus kurzort a dx,dy koordinátáit dx-szel, dy-nal változtatja
<code>moveto(x,y)</code>	A grafikus kurzort az (x,y) pontba mozgatja
<code>x=getx();y=gety()</code>	A grafikus kurzor koordinátái
<code>putpixel(x,y,c)</code>	c színnel rajzol egy képpontot
<code>line(x1,y1,x2,y2)</code>	A két pontot egyenessel köti össze
<code>linereel(dx,dy)</code>	A dx,dy-nal változtatott pontba húz egyenest az aktuális pontból
<code>lineto(x,y)</code>	Az (x,y) pontba húz egyenest az aktuális pontból
<code>drawpoly(n,poly[])</code>	Egy poligont rajzol, ha zárt akkor csúcspontjainak száma n-1, különben n, a poly[] vektorban kell felsorolni a csúcsok koordinátáit, x1,y1,x2,y2,...sorrendben, zárt sokszög esetén az első koordinátáknak egyezniük kell az utolsókkal.
<code>rectangle(x1,y1,x2,y2)</code>	Téglalap x1,y1 bal felső, x2,y2 jobb alsó csúcs koordinátái
<code>arc(x,y,ksz,vsz,r)</code>	Körív, középpontja x,y, sugara r, az ívhosszat ksz-től vsz-ig fokokban.
<code>circle(x,y,r)</code>	Kör az előző paraméterezéssel.
<code>ellipse(x,y,ksz,vsz,rx,ry)</code>	Egy ellipszist, vagy annak egy ívét rajzolja, rx szélességének, ry magasságának fele.
<code>bar(x1,y1,x2,y2)</code>	Kitöltött téglalap.
<code>bar3d(x1,y1,x2,y2,m,t)</code>	Hasáb 45 fokos axonometriában m a mélység, ha t!=0, akkor a fedőlap is festett.
<code>pieslice(x,y,ksz,vsz,r)</code>	Mint az arc, csak ez kitöltött körcikk lesz

¹⁵ Ne feledkezzünk el az útvonal megadásakor a \ helyett \\ használatáról!

<code>floodfill(x,y)</code>	Kitölt egy zárt területet, melynek x,y belső pontja.
<code>cleardevice()</code>	Törli a grafikus képernyőt.
<code>outtext(text)</code>	Szöveget ír a képernyőre az aktuális kurzorpozíciótól
<code>ottextxy(x,y,text)</code>	A szöveget az x,y koordinátájú ponttól kezdi.
<code>settextstyle(f,d,s)</code>	A szöveg stílusát állíthatjuk be vele. f a karakter stílusa <small>DEFAULT_FONT, TRPLEX_FONT, SMALL_FONT, SANS_SERIF_FONT, GOTHIC_FONT</small> d=0 esetén balról jobbra, 1 esetén felülről lefelé ír s a karakter méretét jelenti, ha 1 akkor 8x8 pixel, 2 esetén 16x16 pixel...

41., 42., 43., 44., 45., 46. feladat

2.16 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` direktíva¹⁶ után adhatunk meg. Ezt a direktívát használtuk már korábban konstansok létrehozására.

Az így létrehozott konstansok futásidőben már nem változtathatók¹⁷. 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

2.16.1 Függvényszerű makrók

A definíció általános alakja:

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

A makró hívása:

```
azonosító(argumentumok);
```

Nézzünk 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))

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));
```

¹⁶ A fordítónak szóló utasítás.

¹⁷ A `const` kulcsszóval definiáltokról ez nem mondható el, ezekkel csak annyit kell tenni, hogy egy rájuk irányítunk egy `pointert` és ennek az értékét módosítjuk.


```

printf("%c",NAGY(c));

getch();

}

```

Első látásra föltűnhet, hogy a makró definícióban 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, akkor 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 tokenizálódik¹⁸, vagy pedig az átadott érték szöveggé 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 közé tenni, azt a műveletet a változó neve elé kitett `#` végzi el.

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

47. feladat

¹⁸ Két szintaktikai egységből egyet készítünk. Egy változónévhez hozzáveszünk még néhány karaktert, ezzel a változó nevét módosítjuk.

3 Feladatgyűjtemény

1. feladat

Az első C programról leírtakat vedd össze a Pascalban tanultakkal

2. feladat

Mi lesz a következő programok outputja:

```
#include <stdio.h>

main()
{
    printf(" Első sor \n");
    printf(" Második sor \n");
}

#include <stdio.h>

main()
{
    printf("Első sor");
    printf("Második \n sor \n");
}

#include <stdio.h>

main()
{
    printf("Hello...\n..oh my\n...when do i stop?\n");
}
```

3. feladat

Mi lesz a következő programok outputja:

```
#include <stdio.h>

main()
{
    int sum, v;

    sum = 10;
    v = 15;
    printf("%d\t%d\n", sum, v);
}

#include <stdio.h>

main()
{
    int v1, v2, sum;

    v1 = 35;
    v2 = 18;
    sum = v1 + v2;
    printf("%d és %d összege %d\n", v1, v2, sum);
}
```

4. feladat.

Miért hibásak a következő változónevek?

```
value$sum
exit flag
3lotsofmoney
char
```

5. feladat

Nézzünk egy példát az összes alaptípusú változóra. Mi lesz a következő program outputja?

```
#include < stdio.h >

main()
{
    int    sum;
    float  money;
    char   betu;
    double dupla;

    sum = 10;
    money = 2.21;
    betu = 'A';
    pi = 2.01E6;

    printf("sum = %d\n", sum );
    printf("money = %f\n", money );
    printf("betű = %c\n", betu );
    printf("dupla = %e\n", dupla );
}
```

6. feladat

Írj egy olyan programot, mely egy valós, egy egész és egy karakter típusú változót hoz létre, kezdőérték nem tartalmaz egyik sem. Írasd ki az értéküket!

7. feladat

A következő matematikai kifejezéseket írd át C kifejezésekké:

$$x = \frac{3y + 7(2a - 1)}{y(y - 1)} \quad a = \frac{7x - \frac{3y + 1}{2y + 3}}{\frac{5x - 1}{3 + 4(x - 3)}}$$

8. feladat

Add meg azt a logikai kifejezést, mely akkor igaz, ha

- x 10-nél nagyobb 2-vel és 3-mal nem osztható szám
- y -5 és 5 közé eső szám

9. feladat

```
int x , y ;
x ^= y;
```

```
y ^= x;  
x ^=y;
```

Mi lesz a változók értéke a kiértékelések után?

10. feladat

Írj programot, mely a billentyűzetről beolvas két egész számot, majd eldönti, hogy a nagyobb osztója-e a kisebbnek. Ha igen kiírja, ha nem akkor kiírja a hányadost és az osztási maradékot. A feladat megoldása során ne használj elágazást! (Gondolj a feltételes értékadásra!)

11. feladat

Írj programot mely kiírja az első 20 egész számot és négyzetüket. A feladatot oldd meg mindhárom tanult ciklus segítségével!

12. feladat

Írj programot, mely kiírja a 1-től 10-ig a szorzótáblát táblázatos formában! A feladat megoldásához tetszőleges ciklust használhatsz.

13. feladat

Írj programot, mely 0-tól 90 fokig 5 fokként kiírja a szögek szinuszát és koszinuszát. A megjelenítés táblázatos formában történjen.

Szög szinusz koszinusz

sorrendben. A megfelelő matematikai függvények a $\sin(f)$, $\cos(f)$. A függvények használatához a program fejlécében szükség lesz a

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

állomány beillesztésére is.

14. feladat

Írd be a switch-nél található példaprogramot. Vizsgáld meg, hogy mi történik abban az esetben, ha a break elmarad! Írd át a kiíró részt is switch-esre!

15. feladat

Írd át az alábbi programsorokat switch szerkezetűre!

```
if( betu == 'X' )  
    s = 0;  
else if ( betu == 'Z' )  
    jelzo = 1;  
else if( betu == 'A' )  
    s = 1;  
else  
    printf("Ismeretlen betű -->%c\n", betu );
```

16. feladat

Írj programot, mely bekér három egész számot és eldönti, hogy szerkeszthető-e a számokból, mint szakasz hosszakból háromszög. Ha igen, akkor azt is eldönti, hogy derékszögű-e!

17. feladat



Írj programot, mely bekér három valós számot. Ezeket sorrendben egy másodfokú egyenlet a,b,c együtthatóiként értelmezi. Add is meg az egyenlet megoldásait. A kiírások a következők közül kerüljenek ki: Nincs gyöke, Egy gyöke van $x=ee.tt$, Két gyöke van $x1=ee.tt, x2=ee.tt$. Itt a tt azt jelenti, hogy a gyököket két tizedes pontossággal írassuk ki.

18. feladat



Írd be a következő programot és magyarázd meg a működését!

```
#include <stdio.h>

main()
{
    int szam,s=0;
    printf("Kérek egy számot 1 és 10 között : ");
    scanf("%d", &szam);

    switch (szam)
    {
        default: printf("Hibás megadás: %d\n", szam); break;
        case 10: s+=10;
        case 9: s+=9;
        case 8: s+=8;
        case 7: s+=7;
        case 6: s+=6;
        case 5: s+=5;
        case 4: s+=4;
        case 3: s+=3;
        case 2: s+=2;
        case 1: s+=1; printf("Az s értéke: %d\n", s);break;
    }
}
```

19. feladat

Kérj be billentyűzetről egy pontszámot, majd mondd meg, hogy ehhez hányas érdemjegy tartozik! Maximális pontszám 100, 91-100 jeles(5), 81-90 jó(4), 66-80 közepes (3), 50-65 elégséges(2), 50 alatt elégtelen(1)!

20. feladat



Írjunk programot, mely beolvassa egy tanuló jegyeit, majd kiszámítja az átlagot. A jegyeket tömbben tárolja. A beolvasásnál csak a jó osztályzatokat fogadja el. (gondolj a scanf-nél megadható ellenőrzési lehetőségre) A tantárgyak száma 10. Az átlag kiszámítása után adjuk meg azt is, hogy az egyes osztályzatok hányszor fordultak elő!

21. feladat



Írjunk programot, mely feltölt egy 100 elemű vektort 1 és 1000 közé eső egész számokkal, majd a következő feladatokat látja el:

- kiszámítja az elemek összegét, átlagát
- kiválasztja a legnagyobbat és a legkisebbet
- kiírja az elemeket eredeti sorrendben, majd rendezi és úgy is megjeleníti azokat

Véletlen számok kezeléséhez a következőket kell tudni.

```
#include <stdlib.h>
randomize();
x=rnd() % n
```

Az utóbbi egy véletlen számot ad a $[0, n-1]$ intervallumban. (A `%` itt a maradékos osztást jelenti). Ha nem hívjuk meg a `randomize()` függvényt, akkor a Pascal-hoz hasonlóan itt is mindig ugyanazokat a véletlen számokat fogjuk eredményül kapni.

22. feladat



Dobjunk három dobókockával 100 000-szer. Minden egyes dobásnál adjuk össze a három számot, majd írjuk ki, hogy melyik összeg hányszor fordult elő!

23. feladat



Modellezzük az ötös és a hatos lottó húzását!

24. feladat



Adjuk meg egy 100 elemű 1-100 közötti számokat sorrendben tartalmazó vektor egy permutációját!

25. feladat



Kérjünk be egy szöveget és egy karaktert a billentyűzetről, majd adjuk meg, hogy az adott karakter hányszor fordul elő a szövegben!

26. feladat



Kérjünk be egy szöveget a billentyűzetről és döntsük el, hogy palindroma-e!

27. feladat



Töltsünk föl egy 6 sorból és 10 oszlopból álló mátrixot 200-nál nem nagyobb véletlen számokkal. Majd végezzük el a következő feladatokat:

- Válasszuk ki minden sorából a legnagyobb elemet, majd adjuk meg a maximumok átlagát!
- Számoljuk ki az összes elem átlagát!
- Hány olyan eleme van a mátrixnak, melyek az átlagnál kisebbek, illetve nagyobbak?

28. feladat



Írjunk programot egy a jegyzetben megadott struktúra alkalmazásával. A program véletlenszerű nevekkel töltse fel mind a vezeték, mind a keresztnéveket. A születési idők 1950 és 1970 közé eső véletlen számok legyenek, a fizetések pedig essenek 50 000 és 100 000 Ft közé. A feltöltések után a következő feladatokat végezzük el:

- Adjuk meg az 1960 és 1965 között születetteket!
- Számítsuk ki az átlagfizetést és az átlagéletkort

29. feladat



Írjunk függvényt, mely paraméterként egy karaktert és egy egész értéket kap, és ennyi darabot ír ki egymás után az átadott karakterből!

30. feladat



Írjunk függvényt mely paraméterként egy valós(a) és egy pozitív(n) egész számot kap. Visszatérési értéke pedig: a^n

31. feladat



Írjunk függvényt, mely három átadott egész számot nagyság szerint sorba rendez, a változás a visszaadott értékekben látszik!

32. feladat



Mit fognak kiírni a következő programrészletek printf függvényei?

```
main()
{
    char c = 'Q';
    char *char_pointer = &c;
    printf("%c %c\n", c, *char_pointer);
    c = '/';
    printf("%c %c\n", c, *char_pointer);
    *char_pointer = '(';
    printf("%c %c\n", c, *char_pointer);
}
```

```
main()
{
    int i1, i2, *p1, *p2;

    i1 = 5;
    p1 = &i1;
    i2 = *p1 / 2 + 10;
    p2 = p1;

    printf("i1 = %d, i2 = %d, *p1 = %d, *p2 = %d\n", i1, i2, *p1, *p2);
}
```

```
main()
{
    int count = 10, *temp, sum = 0;

    temp = &count;
```

```

    *temp = 20;
    temp = &sum;
    *temp = count;
    printf("count = %d, *temp = %d, sum = %d\n", count, *temp, sum );
}

```

33. feladat



Írjunk programot, amely véletlenhosszúságú sorozatokat kezel. A sorozat hossza 50 és 200 között változhat. minden sorozat esetén végezze el a következő feladatokat:

- Feltölti a sorozatot véletlenszerű fej/írás értékekkel (F/I, vagy 0/1)
- Meghatározza a fejek és írások számát, arányukat.
- Meghatározza a leghosszabb fej illetve írás sorozat hosszát, és ezek arányait a teljes sorozathoz.

Az adatok mindig csak annyi helyet foglaljanak el a memóriában, amennyire éppen szükség van. Legalább 10 sorozaton végezzük el a fenti három feladatot. Az eredményeket tárolni nem kell, jelenítsük meg azokat táblázatszerűen!

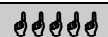
34. feladat



Egy telephelyen gyümölcsöt vásárolnak fel. Minden Eladót a sorszámával azonosítanak. Egy eladó egy hónap folyamán többször is leadott gyümölcsöt, de a különböző eladók nem ugyanannyiszor. Egy vektorban tárolva vannak a havi leadások számai. A telephelyen összesen 10 eladót tartanak nyilván. Írjunk programot, mely beolvassa az eladónként és alkalmanként a leadott gyümölcsmennyiségeket (Lehetnek ezek véletlen számok is 500-100 között), majd adjunk választ a következő kérdésekre:

- Ki adta le a hónapban összesen a legtöbb és a legkevesebb gyümölcsöt?
- Ki adta le egyszerre a legtöbb gyümölcsöt?
- Mennyi gyümölcsöt adtak le egyenként az egyes eladók?

35. feladat



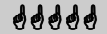
Írjunk programot, mely egy dinamikus listába emberek adatait viszi föl. Név, születési év, lakcím. Majd valósítsuk meg menüből a következő funkciókat:

- egy adat fölvitele a listába
- egy adat törlése a listából
- keresés név alapján (egyedi jellemző)
- keresés születési év alapján (több egyforma is lehet, mindet adja meg)
- teljes lista

36. feladat



Írjunk programot, amely megszámolja hogy egy adott szöveg hány szóból áll. Vigyázzunk arra, hogy a szóközök száma+1 nem feltétlenül a szavak számát adja vissza, előfordulhat egy szövegen belül több szóköz is. A szöveget egy állományból vegyük! Lehet ez például egy programállomány is.

37. feladat

Készítsünk a 35. feladathoz hasonló programot, az adatokat viszont ebben az esetben nem dinamikus listába, hanem egy bináris állományba kell fölvennünk!

38. feladat

Készítsünk programot, mely bekéri egy állomány nevét, ha létezik ilyen állomány, akkor azt karakterenként végigolvassa és minden karakter nagybetűs alakját kiírja egy másik állományba, melynek a nevét az elsőből úgy kapjuk, hogy egy n betűt hozzáírunk. A kiírás után a nagybetűs állományt jelenítse is meg a képernyőn!

39. feladat

Készítsünk ablakozó technikát használó programot, a képernyőtartalmat elmentő függvények segítségével! Ablakozó technikát figyelhetünk meg például a karakteres képernyőn az NC program használatakor. Ez azt jelenti, hogy ha egy menüből kiválasztunk egy pontot, akkor a pontosítás érdekében egy almenü jelenik meg, mely eltakarja az alatta lévő tartalmat, ha ez az ablak eltűnik, akkor vissza kell tölteni azt a részt, ami alatta volt.

40. feladat

Készítsünk rajzoló programot. A szöveges képernyőn a kurzort a kurzormozgató nyilakkal lehessen mozgatni, legyen benne egy penup(u) és egy pendown(d) funkció. A nyilak kódja: fel 72, jobbra 77, balra 75, le 80. Vigyázzunk ezeknek a beolvasásával, ugyanis ezek a billentyűk kettős kóddal rendelkeznek, tehát a lenyomáskor két érték kerül a billentyűzetpufferbe, az első kód minden esetben 0, a második pedig az imént felsorolt. Az ilyen billentyűket úgy szoktuk földolgozni, hogy ha az első karakter 0, akkor olvasunk még egyet a pufferből. Ezeket az olvasásokat a getch() függvénnyel tehetjük meg.

41. feladat

Írjunk függvényt mely grafikus módban kirajzolja az olimpiai öt karikát. A függvényt a körök sugarával és a felső sor középső karikájának középpontjával lehessen meghívni!

42. feladat

Írjunk programot, mely kirajzol egy sakktáblát a képernyőre. Legyen mód arra, hogy a kitöltés mintáját változtassuk!

43. feladat

Írjunk programot, mely vektorban tárolt számokat megjeleníti kör és oszlop diagramos formában is. Kördiagram esetén a mennyiségek összege fele meg 360 foknak. A szomszédos körökkelnél alkalmazzunk eltérő kitöltést! Oszlopdiagram esetén a legnagyobb mennyiség töltsen ki függőlegesen a képernyő két harmad részét, természetesen a többi mennyiség ehhez képest arányos magasságú legyen!

44. feladat

Írjunk egy olyan programot, mely maximum ötödfokú polinomokat tud ábrázolni. Az ötödfokú polinomok általános alakja: $a_5 \cdot x^5 + a_4 \cdot x^4 + a_3 \cdot x^3 + a_2 \cdot x^2 + a_1 \cdot x + a_0$. A programban a megfelelő együtthatókat kérjük be, illetve az a változó intervallumának alsó és felső határát. Vigyázzunk az ábrázoláskor, hogy az adott intervallumon minden függvényérték férjen el a képernyőn!

45. feladat

Készítsünk egy céltáblát!

46. feladat

Készítsünk programot, mely a következő paraméteres egyenlettel megadott görbét ábrázolja:

$$x = a \cdot \sin(n \cdot \varphi), \quad y = b \cdot \cos(m \cdot \varphi)$$

A φ paraméter értéke a $[0, k\pi]$ intervallumon változzon. Az összefüggésben szereplő változók közül a, b valós a többi egész típus. Lehetőleg kis egészekkel dolgozzunk.

47. feladat

Írjunk makrókat a következő funkciókra:

- Két szám közül adja vissza a nagyobbát.
- Ha egy karakter nagybetű, akkor adja vissza a kisbetűs megfelelőjét.
- Adja vissza egy szám négyzetét.

4 Függelék