TÁMOP-4.1.1.F-13/1-2013-0010

„Eltérő utak a sikeres élethez!”A Miskolci Egyetem társadalmi, gazdasági szerepének fejlesztése, különös tekintettel a duális képzési típusú megoldásokra

PROGRAMOZÁS ALAPJAI (ANSI C NYELVEN)

Mérnök informatikus duális képzést támogató oktatási anyag

Összeállította:
Dr. Baksáné dr. Varga Erika
Dr. Hornyák Olivér

Gépészmérnöki és Informatikai Kar
Informatikai Intézet

MISKOLCI EGYETEM

2015.

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE

TARTALOM

1 Bevezetés

Ez a jegyzet a hagyományos és az úgynevezett duális képzésben egyaránt használható a C nyelvű programozás alapjainak gyakorlati oktatásában. Az oktatási anyag összeállításakor fő szempont volt, hogy segítséget nyújtsunk a duális képzésben gyakorlati órákat adó informatikus kollégák munkájához. Nem volt könnyű dolgunk a feladatok kiválasztásakor, mert egyszerre kellett szem előtt tartanunk a hallgatók tapasztalatlanságát és a gyakorló informatikus kollégák jártasságát. Ezért a jegyzetben keveredik a didaktikus szemlélet a gyakorlati megfontolásokkal. Másrészt a témakörök meghatározásakor igazodnunk kellett az elméleti tananyag menetéhez. Ilyen szempontok figyelembevételével készült el ez a jegyzet. Ebben a szerzők igyekeznek megismertetni a hallgatókat a programozás alapelveivel és módszereivel gyakorlati példák révén, miközben segítséget nyújtanak a gyakorlattal rendelkező informatikusok számára, hogy rendszerezhessék, és ezáltal hatékonyabban adhassák át tudásukat a fiatalabb generációnak.

Elvileg bármelyik programozási nyelv alkalmas arra, hogy a programozás alapjait megtanítsuk általa. Mi erre a célra az ANSI C nyelvet választottuk. Egyrészt mert a Miskolci Egyetem mérnök-, programozó- és gazdaságinformatikus alapszakokon ezt oktatjuk, másrészt mert sok más nyelvnek ez az alapja. Amellett, hogy jó kiindulópont az imperatív (azaz a számítógépet az egymás után végrehajtandó utasítások megadásával manipuláló) programozási paradigma megértéséhez, több programtervezési alapelv szemléltetésére is kiválóan alkalmas. Procedurális mivoltából következően egy programozási feladat megoldása egymástól többé kevésbé független, jól definiált alprogramból (*procedure*) épül fel; és jól használható strukturált és moduláris programok fejlesztéséhez. Arra biztatjuk a kedves Olvasót, hogy a tananyag feldolgozása után ne álljon meg ott, ahol a jegyzet véget ér: vágjon bele egy objektum-orientált programozási nyelv elsajátításába, és használja fel az itt megtanult alapokat!

A 2.-6. és a 13. fejezeteket Dr. Hornyák Olivér, egyetemi docens írta. A 7.-12. és a 14. fejezeteket Dr. Baksáné dr. Varga Erika, egyetemi adjunktus állította össze.

“*Advice to students*: Leap in and try things. If you succeed, you can have enormous influence. If you fail, you have still learned something, and your next attempt is sure to be better for it.”

(Brian Kernighan, 2009)

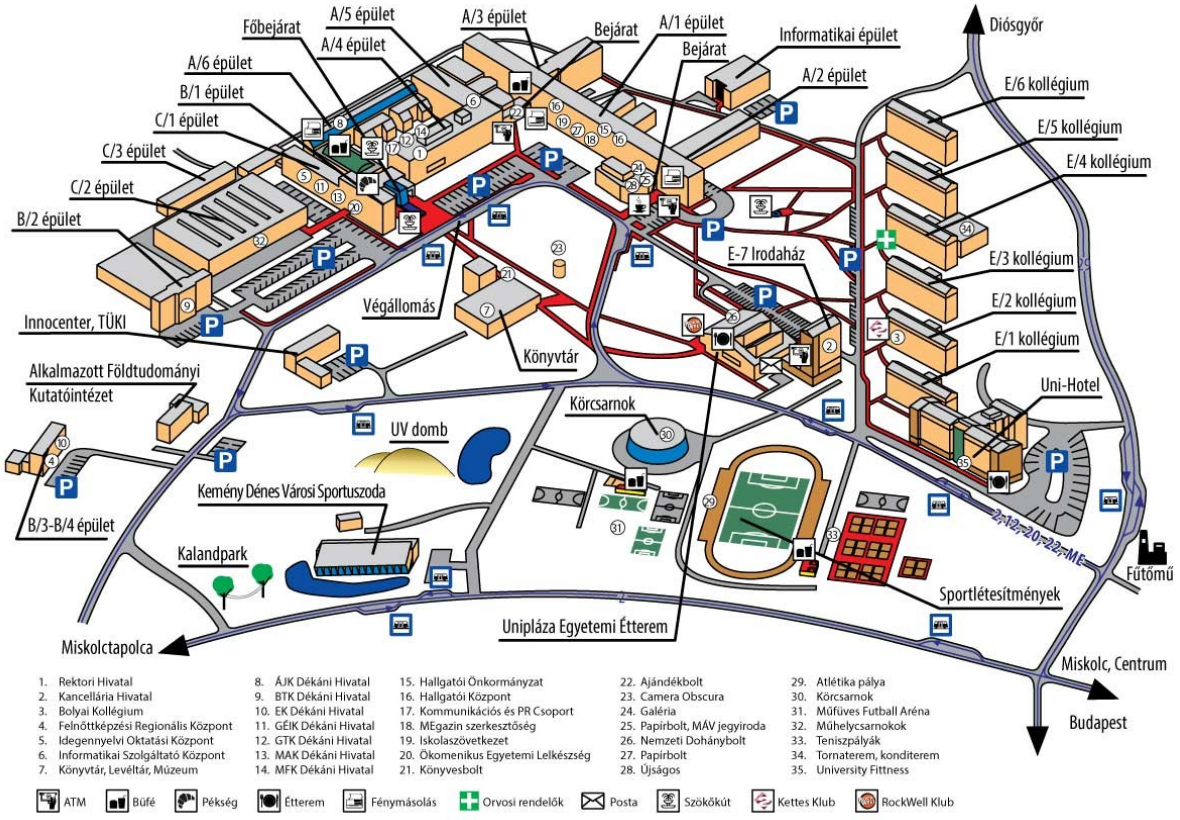
Sok sikert kívánunk a programozás alapjainak elsajátításához!

2 Algoritmusok

A számítástudomány (computer science) fogalmát elég nehéz definiálni. Benne van a nevében a számítás, pedig nagyon gyakran nem is valamilyen számértékkel foglalkozunk, amikor számítógépekkel dolgozunk.
A számítástudomány a problémák tanulmányozásának, a problémák megoldásának a művészete számítógépekkel. Adott egy probléma, és keressük azoknak a lépéseknek a sorozatát, amellyel a problémát megoldhatjuk. Ezt nevezzük *algoritmusnak*.
Algoritmus: lépések véges sorozata, amelyeket követve megoldhatjuk a problémát.
A számítástechnikát megtanulni csak gyakorlati tapasztalaton keresztül lehet. Mi is úgy tanulunk, hogy mások problémáit megnézzük, és megpróbáljuk megoldani. Természetesen többféle megoldás – azaz több helyes algoritmus – is lehetséges. Ha egy csomó feladatot már megoldottunk, egy csomó algoritmust kidolgoztunk, akkor észrevehetjük a bennük rejlő hasonlóságokat. És ha legközelebb egy problémába ütközünk, akkor máris egy hasonló megoldó algoritmusból indulhatunk ki.
Az algoritmusok nagyon különbözőek. Van, amelyik erőforrásigénye kisebb, van, amelyiké nagyobb. Van, amelyik ugyanazt a feladatot oldja meg, mégis tízszer, százszor lassabb, mint a másik. Ha eljutsz az algoritmizálásban odáig, hogy analizálni is tudod az algoritmusaidat, akkor igen nagy lépést tettél, hogy IGAZI PROGRAMOZÓ legyél.
Lássunk egy példát:
A Miskolci Egyetem Informatikai Intézetében vagy és szeretnél eljutni a Menzára gyalogosan. Egy térképet is találsz, amely segít megtervezni az útvonaladat. Ez máris egy absztrakció: a térkép a valóságnak egy szimbolikus leképzése. Ha kijutottál az intézeti épületből, menj balra az E/2 kollégiumig. Ott fordulj jobbra, és menj tovább a Menzáig. Ez egy algoritmus volt. Leírtuk, merre menj. Van másik lehetséges algoritmus is: Az épületből kijutva menj balra az első sarokig. Ott fordulj jobbra, majd a tanulmányi épületeknél balra. Ez egy másik algoritmus. A bejárt út is rövidebb. Talán nehezebb követni, de hatékonyabb. Mehetnél akár jobbra is, megkerülve a főépületet, a műhelycsarnok épületénél balra fordulva. Láthatod: egy egyszerű feladatnál is több megoldó algoritmus lehetséges.
Vannak olyan algoritmikus lépesek, amelyek ismétlődnek. Például, ha kiléptél az épületből, akkor fordulj balra. Lépj egyet, majd még egyet, majd, még egyet…így tovább, ismételd meg a lépéseket jónéhányszor. Ezeket ciklusoknak nevezzük. Ez egy olyan algoritmus, amiben számoljuk a lépések számát, és megadott darabszámú iterációt (ismétlést hajtunk végre).
De olyan ciklusok is vannak, amikor minden lépés után eldöntöm, folytatom-e a ciklust. Például:

1. Fordulj balra!
2. Amíg nem érted el az E/2 épületet
    - Lépj előre!

3. Fordulj jobbra!
4. Amíg nem érted el a Menzát
    - Lépj előre!



1. ábra A Miskolci Egyetem térképe (Forrás: www.uni-miskolc.hu)

2.1 Folyamatábra

Az algoritmusokat le lehet írni szövegesen is, eddig ezt tettük. Ezzel készíthetünk pszeudo-kódot is. Ez egyfajta meta kód. Az algoritmust folyamatábrával szokás ábrázolni. A folyamatábra szimbólumok a következők:

2.1.1 Kezdet/vég

Ezzel az alakzattal jelölheti a folyamat első és utolsó lépését.

2. ábra Kezdő és vég alakzat

2.1.2 Folyamat

Ez az alakzat a folyamat egy lépését jelöli.

3. ábra Folyamat

2.1.3 Elágazás

Ez az alakzat egy döntési helyzetet jelöl. A kiértékeléstől függően vagy így, vagy úgy folytatódik.

4. ábra Elágazás

2.1.4 Adatok

Ez az alakzat azt jelzi, hogy kívülről adatok érkeznek a folyamatba, vagy adatok kerülnek ki a folyamatból. Az alakzat anyagokat is jelölhet, és Bemenet/kimenet alakzat néven is ismert.

5. ábra Adatok

2.1.5 Hivatkozás

Ez a kis kör azt jelzi, hogy a következő (vagy az előző) lépés másutt található a rajzon. Ez különösen a nagyméretű folyamatábráknál hasznos, amelyeken máskülönben hosszú, nehezen végigkövethető nyilakat kellene alkalmazni.

6. ábra Hivatkozás

2.1.6 Következő lépés

A nyíl irányába haladva mutatjuk a következő lépést

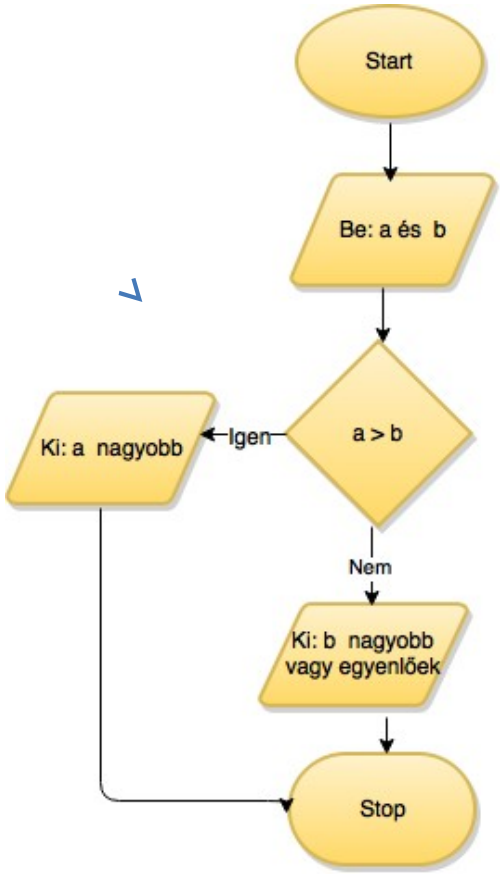
7. ábra Követező lépés

2.2 Példák algoritmusokra

2.2.1 Elágazás

Melyik szám a nagyobb?

1. Kérjük be a és b értékét
2. Ha a > b, írjuk ki, hogy az a nagyobb
3. Egyébként írjuk ki hogy
   b a nagyobb, vagy egyenlőek

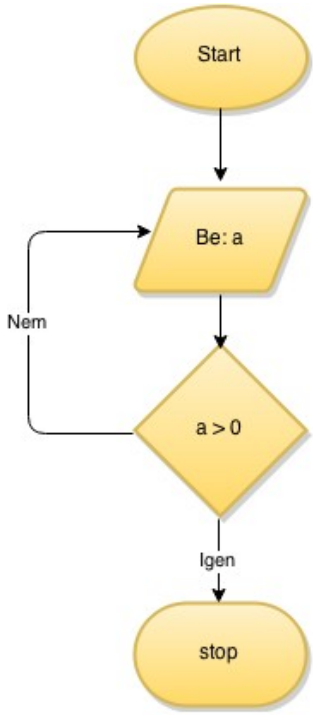


8. ábra Elágazás

2.2.2 Ciklus

Kérünk be egy pozitív egész számot. Hibás bevitelnél ismételjük meg a bekérést!

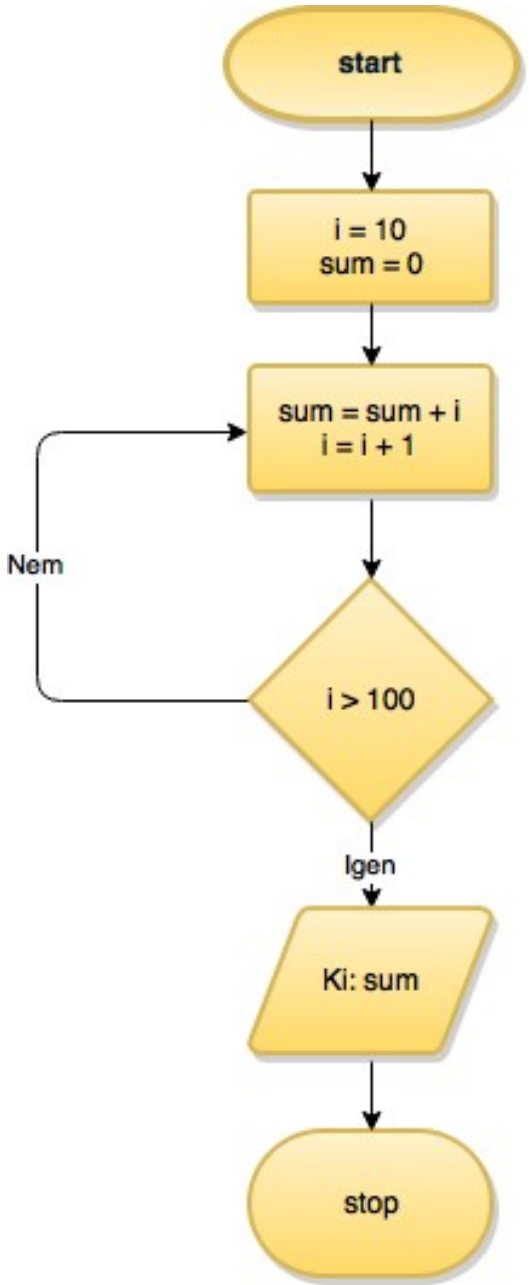
1. Bekérjük *a* értékét
2. Ha *a* nagyobb nullánál, akkor végeztünk
3. Egyébként vissza a bekéréshez



9. ábra Pozitív szám bekérése

2.2.3 Összegzés

Adjuk össze a pozitív egész számokat tíztől százig!
1. *i* értéke legyen tíz, *sum* értéke legyen nulla
2. *sum* értékét növeljük meg *i* értékével
3. *i* értékét növeljük meg eggyel
4. Ha *i* nagyobb, mint száz végeztünk (tehát a százat még hozzáadjuk a *sum* változóhoz, de százegyet már nem)
5. Egyébként vissza a 2. lépéshez
6. Kiírjuk *sum* értéklt

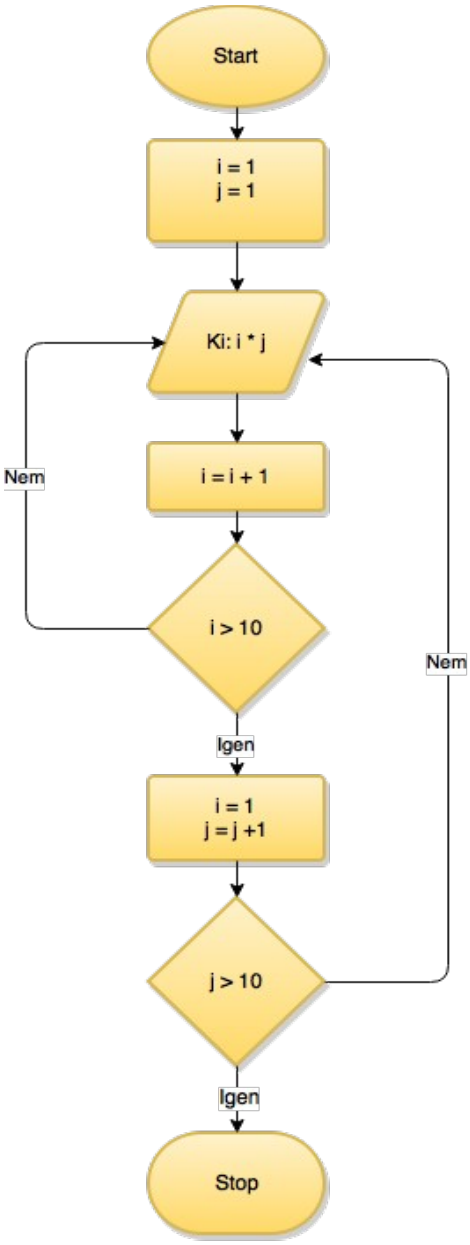


10. ábra Összeadás tíztől százig

2.2.4 Tízes szorzótábla

Írassuk ki a tízes szorzótáblát!

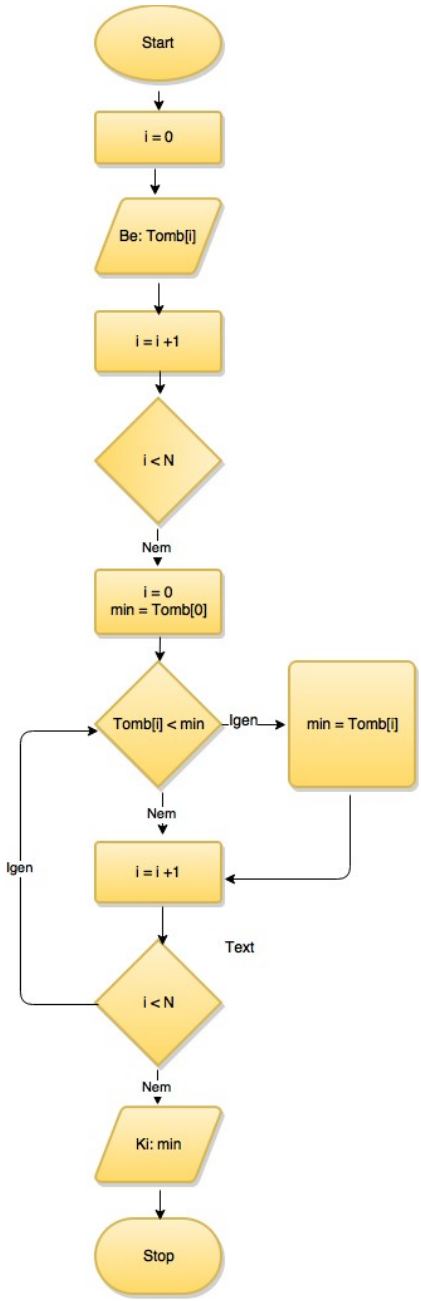
Ehhez beágyazott ciklusokra van szükség.
1. *i* és *j* értéke legyen 1
2. A belső ciklus ciklusváltozója *i.*
Kiírjuk az i*j szorzatot
3. *i* értékét növeljük eggyel.
4. amíg *i* értéke kisebb tíznél
5. Amikor *i* értéke 10-ről 11-re növekszik, akkor az *i* > 10 feltétel igaz lesz.
6. Ekkor *i* értékének újra 1-et adunk és a külső ciklus ciklusváltozója *j* értékét is növeljük.
7. Ha *j* a külső ciklusváltozó nagyobb, mint tíz, akkor vége
8 Egyébként vissza a 2. ponthoz



11. ábra Tízes szorzótábla

2.2.5 Vektor legkisebb eleme

1. A Start után *i* értékét lenullázzuk.
2. Bekérjük az egydimenziós *Tomb* változó *i*. elemét
(Figyelem, a tömbök, vektorok elő eleme a 0 indexet kapja!)
3. Ezután i értékéhez 1-et adunk.
4 Amíg N értékét el nem érjük, addig ezt ismételjük.
5. A legkisebb értéket a *min* változó fogja tartalmazni, ennek kezdőértéke legyen a *Tomb* első eleme, amelynek indexe 0.
6. Ezután az *i* ciklusváltozó segítségével megvizsgáljuk a *Tomb* összes elemét.
i értékel legyen 0.
7. Ha a *Tomb i*. eleme kisebb, mint a min változó értéke, akkor ez lesz az eddig legkisebb érték.
8. Ezért *min* értéke vegye fel a *Tomb i.* elemének értékét.
9. Ezután i értékét növeljük 1-el.
10. Ismételjük, amíg az N. elemet is meg nem vizsgáltuk.
11. Ha az N. elemet megvizsgáltuk, akkor vége.



12. ábra Vektor minimumkeresés

2.2.6 Sorbarendezés

1. A Start után i értékét lenullázzuk.
2. Bekérjük az egydimenziós *Tomb* változó *i*. elemét
3. Ezután *i* értékéhet 1-et adunk
4. Amíg N értékét el nem érjük, addig ezt ismételjük.

Magához a sorba rendezéshez egymásba ágyazott ciklusokra van szükség.

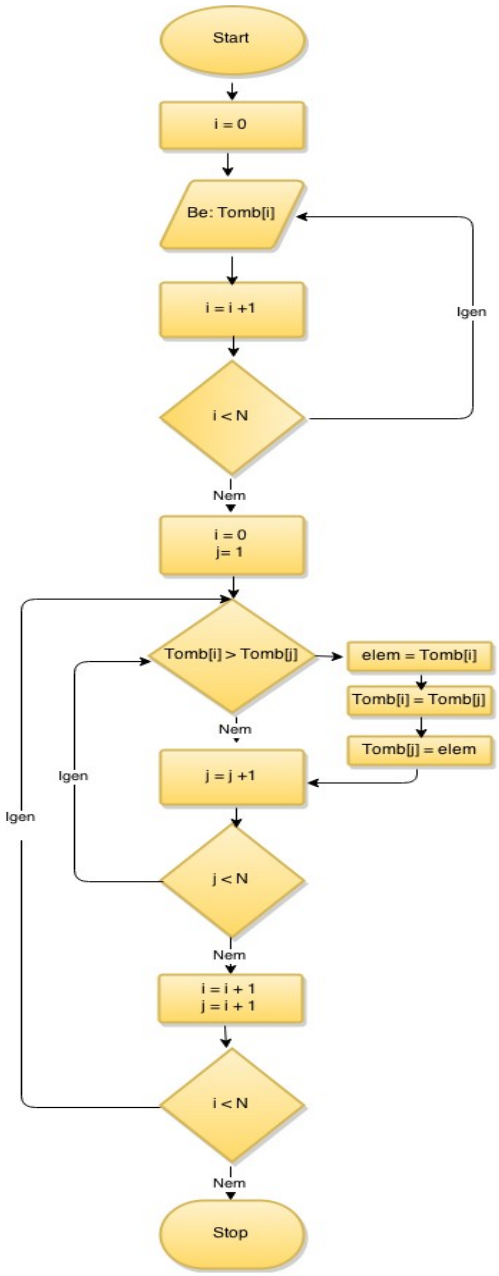
5. A belső ciklus ciklusváltozója *j*. Lenullázzuk. A belső ciklusváltozó, *i* kezdőértéke 1 lesz
6. A belső ciklus segítségével a Tomb *i.* elemét összehasonlítjuk az összes Tomb elemmel, amelynek az indexe nagyobb, mint *i,* azaz a *j* ciklus *i*+1-től fut amíg el nem érjük az N elemszámot
6. Ha Tomb i. eleme nagyobb, mint a j. akkor
7. megcseréljük a két elemet.

7/1. A csere 3 lépésben történik: az *elem* változó eltárolja az i. értéket.

7/2. Ezután az *i*. érték helyére beírjuk a *j*. értéket.

7/3. Majd a *j*. érték helyére azt, amit az *elem* változó tárol.

8. A külső ciklus ciklusváltozója *i* és a ciklus *i* = 0 értéktől addig fut,
9. *i* és *j* érékét egyesével növelve
10. amíg N értékét el nem érjük.

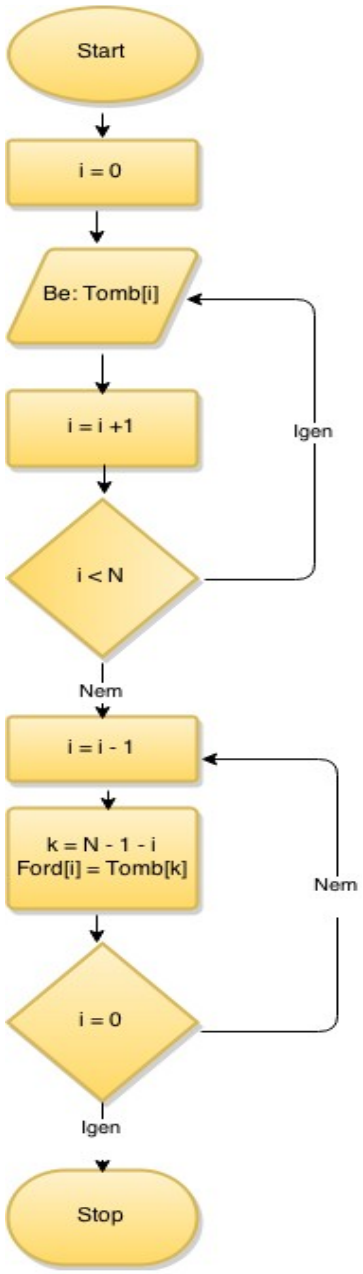


13. ábra Sorbarendezés

2.2.7 Fordított sorrend

Egy N elemű vektorból képezzük azt a vektort, amely az elemeket fordított sorrendben tartalmazza.

1. A Start után i értékét lenullázzuk.
2. Bekérjük az egydimenziós Tomb változó *i*. elemét
3. Ezután *i* értékéhet 1-et adunk
4. Amíg N értékét el nem érjük, addig ezt ismételjük.
5.Ezután az i ciklusváltozóval visszafelé fogunk haladni. Addig vonunk ki belőle egyet, amíg az első – 0. indexű – elemmel nem végeztünk.
6. A ciklus magjában a Ford vektor i. eleme az eredeti, Tomb nevű vektor k. elemével lesz egyenlő.
k értékét az N-1-i összefüggéssel számolhatjuk ki.
Például, ha a vektorunk N=4 elemű, akkor az i = 0, 1, 2, 3 értékek esetén k rendre 3, 2, 1, 0 értékeket veszi fel.
Mivel itt a fordított sorrendet egy Ford nevezetű vektorban tároljuk, ezért az eredeti, Tomb vektorunk megmarad változatlan formában
7. Elértük a 0 értéket ? Ha nem, vissza a 5 ponthoz
8. Ha igen, végeztünk



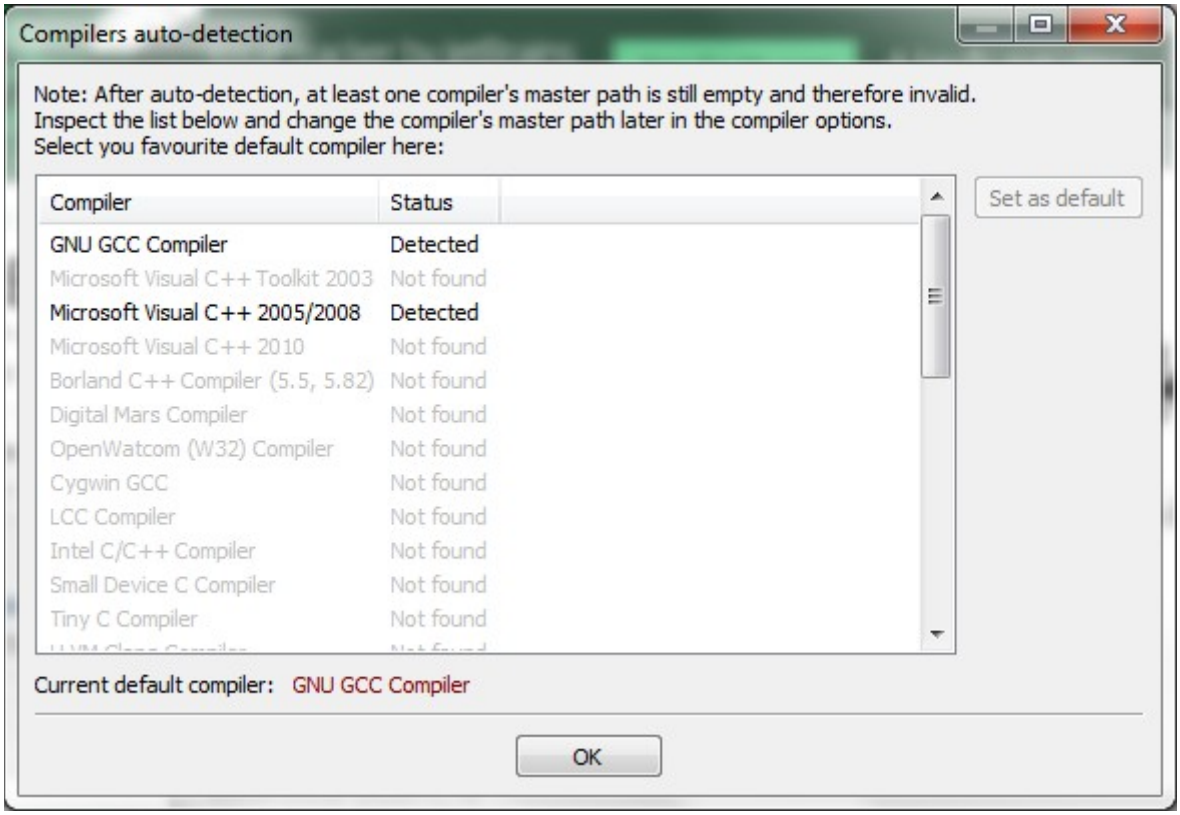
14. ábra Fordított sorrend

3 A Code::Blocks használata

A Code::Blocks aze egy integrált fejlesztőkörnyezet (Integrated Developer Environment, IDE). Ebben a fejezetben végigvesszük a kezdő lépéseket, amelyekkel egy C nyelvű kódot futtatni tudunk a fejlesztőkörnyezetben. A CodeBlock letöltését és telepítését nem írjuk le, a jegyzet megírásakor a telepítőcsomag az alábbi linken érhető el:
http://www.codeblocks.org/downloads
A telepítést varázsló vezeti végig.

3.1 Code::Blocks futtatása

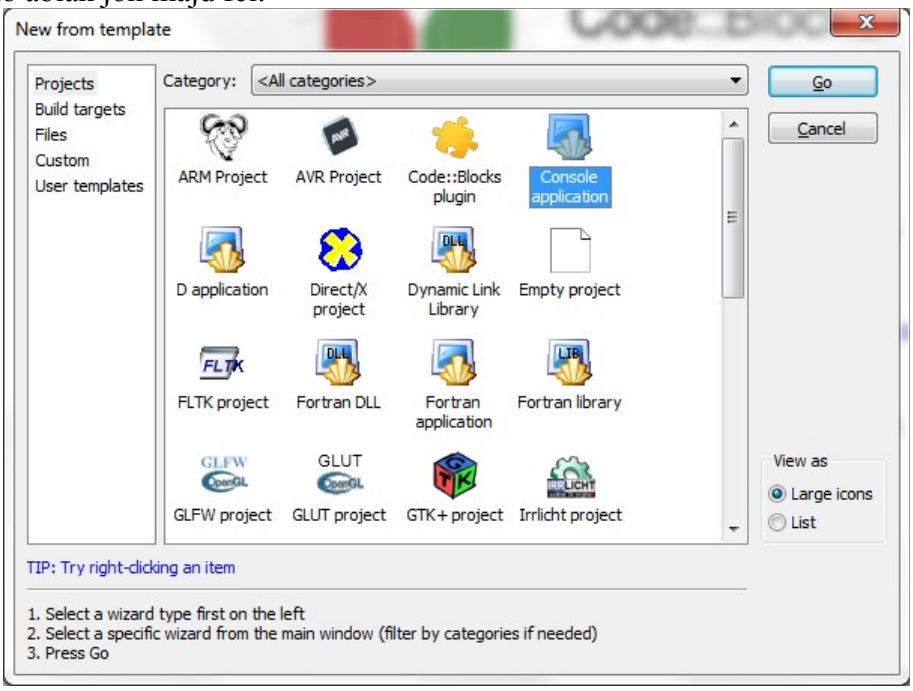
Jelöljük ki, melyik C fordítót használjuk a rendelkezésre álló fordítók listájából. Ebben a segédletben ANSI C nyelvű kódot találsz, azt mindegyik fordítónak támogatnia kell.



15. ábra Compiler (fordító) kiválasztása

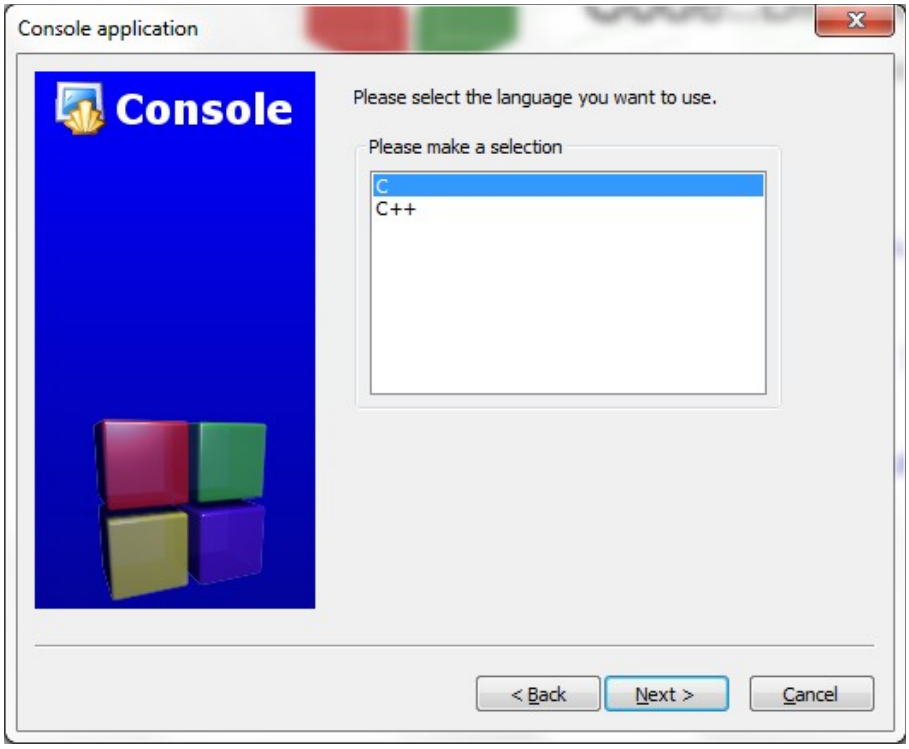
Miután kiválasztottuk a fordítót, nyomjunk OK gombot. Ha a Code::Blocks megkérdezi, akarjuk-e a programot az alapértelmezett C/C++ fáj nézegetőként megjelölni, döntsük el. Majd File menü, New menüpontjában válasszuk, hogy"Project..."

A következő ablak jön majd fel:



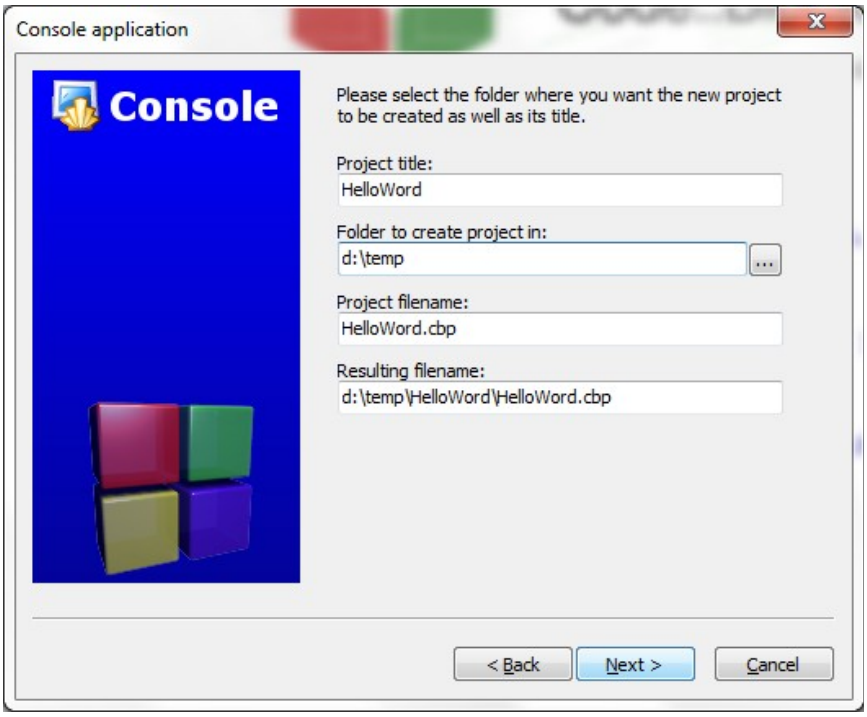
16. ábra Új projekt létrehozása

Válaszd a következőt: "Console Application" majd a "Go" gomb. Azután Next gomb.
A konzol applikáció olyan alkalmazás, amely a szöveges képernyőre – a konzolra – ír. Ebben a segédletben főleg ilyen programokat találsz.



17. ábra C nyelv kiválasztása

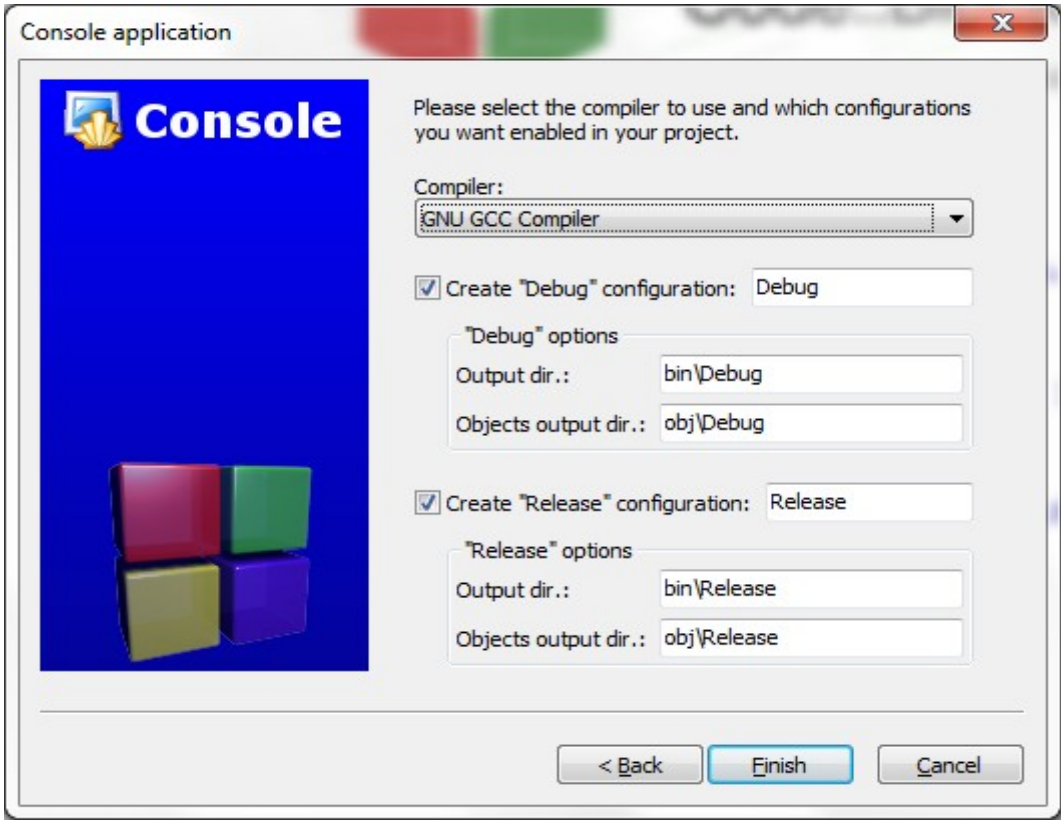
Válaszd a C nyelvet. A Next gomb után adjunk projekt nevet és könyvtárnevet, valahogy így:



18. ábra Projekt elnevezése

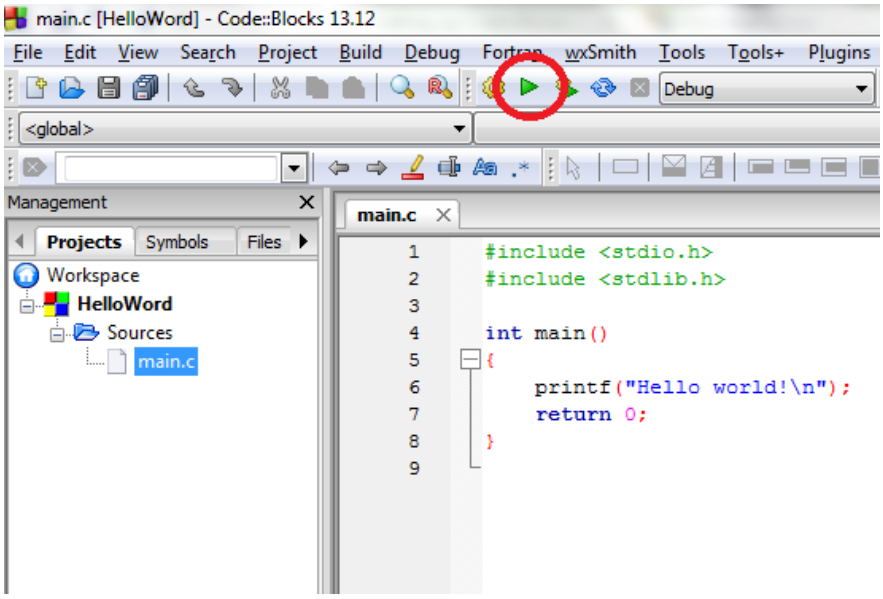
A projekt összefogja az alkalmazásodhoz készített összes fájlt. Bonyolultabb alkalmazások több fájlból is állhatnak. Célszerű ezért az összetartozó projekt fájlokat egyazon könyvtárban elhelyezni. Ebben a dialógus ablakban meg kellett adni a projekt nevét és azt a könyvtárat, ahol a fájlok keletkeznek. Maga a projekt is igényel egy projektleíró fájlt. Ennek a kiterjesztése a cbp (CodeBlocks Projekt).
A Next gomb megnyomása után kiválaszthatjuk, hogy a projektünket milyen konfigurációban akarjuk engedélyezni. A kódolás folyamata során a Debug mód (hibakeresés) erősen javasolt. Ilyenkor a fordító olyan információkat is elhelyez a kódban, amelynek segítségével a változók aktuális értékeit, a függvények híváslistáját és még sok egyéb dolgot is le tudunk kérdezni.
Amikor a kódunk kész, és futtatható állományt készítünk, hogy kibocsássuk az alkalmazásunkat, akkor erre nincs szükség, ilyenkor a Release (kibocsájtás) konfiguráció javasolt.
A következő dialógus ablakban választhatunk a rendelkezésre álló fordítók közül, és engedélyezhetjük a Debug illetve a Release módot. A C fordító két lépésben dolgozik: először a *compiler* egy úgynevezett object fájlt készít a forráskódunkból. Aztán az object fájlokat a

linker fűzi össze futtatható állománnyá. A dialógus ablakban az obj kiterjesztésű object fájlok könyvtárát illetve az összefűzött fájl kimeneti (output) könyvtárát adhatjuk meg.



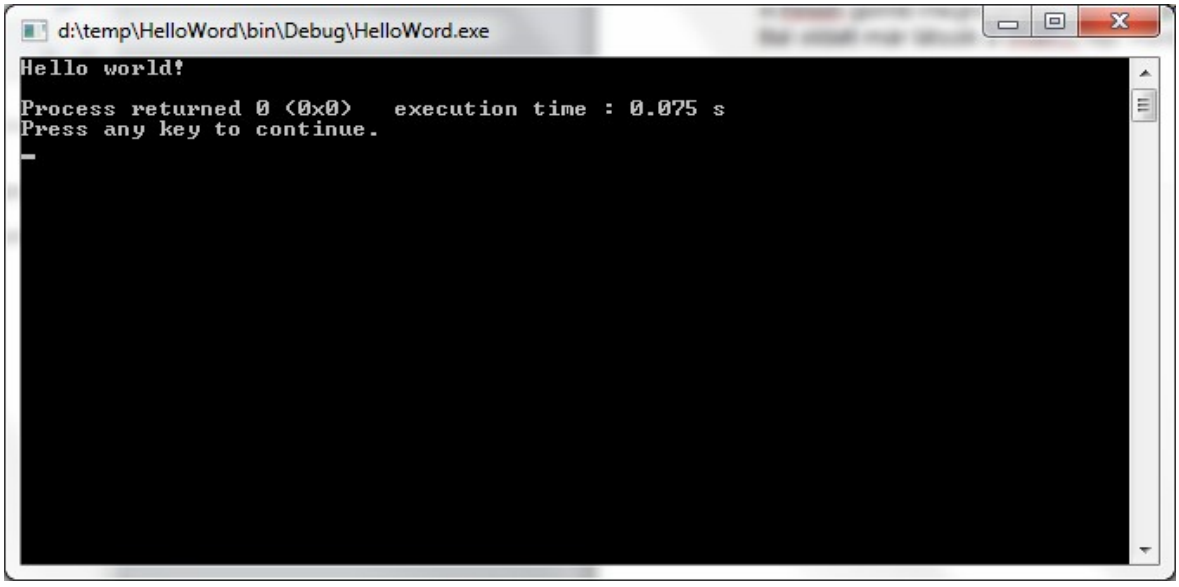
19. ábra Konfiguráció

A Finish gomb megnyomása után megkapjuk a projekt ablakot. Bal oldalt már látszik a main.c fájl. Fordítsuk le a bekarikázott ikon segítségével



20. ábra main.c fájl

És íme a futási eredmény:



21. ábra Az első futási eredmény

3.2 A C kód felépítése

Az első C programunkat meg is írtuk

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

int main()
{
printf("Hello, World\n");
return 0;
}
```

Így nézett ki. A C programunk függvényekből áll. A legfontosabb a main nevű függvény. Ez a programunk végrehajtásának kezdőpontja. A kapcsos zárójelek között van a kód. A függvényeknek rendszerint van visszatérési értéke. A mi main függvényünk egy egész számot (int) fog visszaadni és nincs paramétere. A legutolsó utasítás a return 0. A return utasítás mindig kilép a függvényből. És miért 0 az értétke? Mert jobb nem jutott az eszembe. A 0 az egy egész szám, a visszatérési értéknek most nincs semmi jelentése.
a printf utasítás az idézőjelek közé írt szövegkonstansot kiírja a képenyőre. A \n az egy speciális karakter – igen, egy karakter – amely soremelést végez el.
A printf függvényt nem definiáltuk, azaz nem mondtuk meg, milyen paraméterekkel kell meghívni, milyen visszatérési értékekkel rendelkezik. Ezt egy másik fájlban, az stdio.h nevezetű fájlban tették meg. Hogy használni tudjuk ezt a függvényt, ahhoz az
`#include <stdio.h>`
paranccsal a fájt preprocesszor hozzáadja a kódunkhoz. A másik include fájlt ez a kód nem használja, de gyakorlatilag minden épkézláb C kódunk használni fogja, ezért nem töröltük ki.

3.3 Hibakeresés

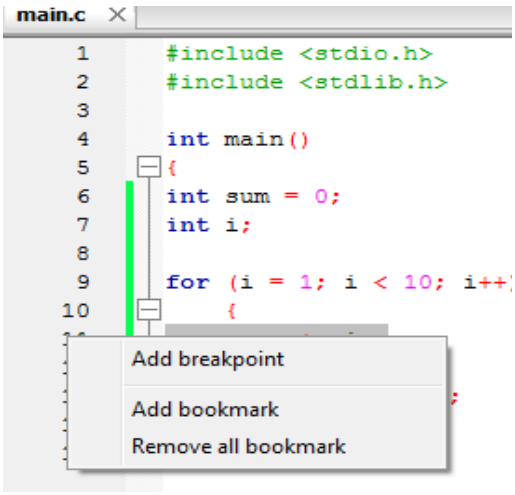
Hogyan tudjuk a kódot lépésenként végrehajtani? Ez a hibakereséskor elengedhetetlen. Legyen a következő kódunk

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

int main()
{
int sum = 0;
int i;
```

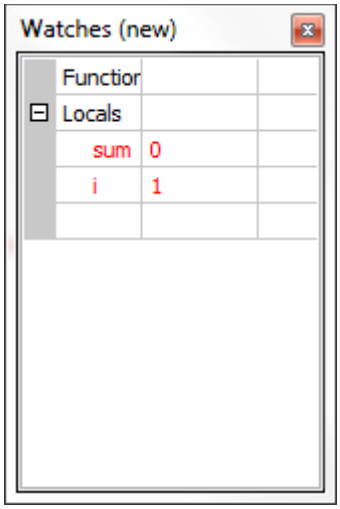
```
for (i = 1; i < 10; i++)
    {
     sum += i;
    }
printf("%d\n", sum);
}
```

Szeretnénk megnézni, hogyan alakul a sum értéke az iterációk során A jobb egérgombbal válasszuk ki annak a sornak a sorszámát, ahol a kód futtatását meg akarjuk állítani és válasszuk az Insert breakpoint (Töréspont beillesztése) opciót. Ha a projektünk Debug módban van, akkor futása ennél a töréspontnál megáll.



22. ábra Töréspont hozzáadása

A Debug menüpontban a Start/Continue pontot választva elindul a hibakeresés folyamata és az első töréspontnál megáll a futtatás.
A Debug menüpont Debugging Window | Watches pipát kiválasztva megjelenik a nézőke, ahol a 2 változónk *(i* és *sum*) értéke megfigyelhető. Folytassuk a hibakeresést az F8 gomb nyomogatásával.



23. ábra Változó értékének nézőkéje

4 Operátorok

A C nyelv operátorait a következő csoportokba sorolhatjuk

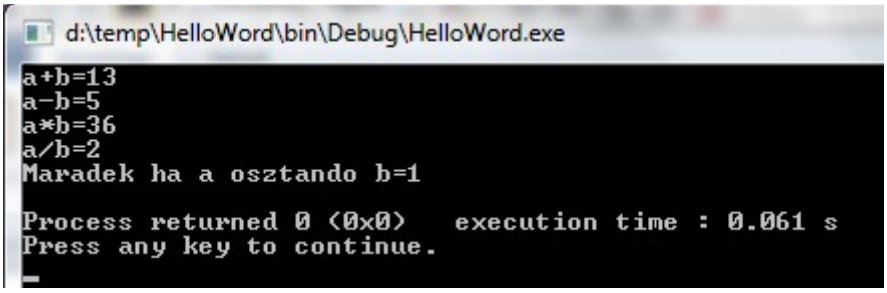
- Aritmetikai operátorok
- Összehasonlító operátorok
- Logikai operátorok
- Bit operátorok
- Értékadás operátorok
- Egyéb operátorok

4.1 Aritmetikai operátorok

Operátor	Jelentés
+	Két szám összege
-	Két szám különbsége
*	Két szám szorzata
/	Két szám hányadosa
%	Modulus (osztási maradék). Csak integer típussal használható
++	Megnöveli a szám értékét eggyel
--	Csökkenti a számot eggyel

Például vegyük a következő kódot. A példa magáért beszél, bemutatja az összeadás, kivonás szorzás osztást.

```
#include <stdio.h>
int main(){
	int a=9,b=4,c;
	c=a+b;
	printf("a+b=%d\n",c);
	c=a-b;
	printf("a-b=%d\n",c);
	c=a*b;
	printf("a*b=%d\n",c);
	c=a/b;
	printf("a/b=%d\n",c);
	c=a%b;
	printf("Maradek ha a osztando b=%d\n",c);
	return 0;
}
```



24. ábra Aritmetikai operátorok

```
/* Legyen a=5 and b=10 */
a++;  //a eredmenye 6
a--;  //a eredmenye 5
++a;  //a eredmenye 6
--a;  //a eredmenye 5
```

A következő kódrészlet bemutatja a ++ és a – operátorok prefix és postfix formátumú operátorok használatát.

```
#include <stdio.h>
int main(){
      int c=2,d=2;
      /* Eloszor kiirjuk a 2-t aztán
    c értékét növeljük eggyel, azaz 3-ra */
    printf("%d\n",c++);
    /* c értékét növeljük eggyel 4-re,
    majd kiírjuk */
      printf("%d",++c);
      return 0;
      }
```

A kiirt értékek:
2
4
l

4.2 Összehasonlító operátorok

Operátor	Jelentés
==	Igaz, ha a két operandus egyenlő
!=	Igaz, ha a két operandus különböző

<	Igaz, ha a bal operandus kisebb, mint a jobb operandus
>	Igaz, ha a bal operandus nagyobb, mint a jobb operandus
<=	Igaz, ha a bal operandus kisebb vagy egyenlő, mint a jobb operandus
>=	Igaz, ha a bal operandus nagyobb, mint a jobb operandus

```
#include <stdio.h>

int main()
	{
	int a = 21;
	int b = 10;

	if( a == b )
		{
		printf("1. sor - a egyenlo b\n" );
		}
	else
		{
		printf("1. sor - a nem egyenlo b\n" );
		}
	if ( a < b )
		{
		printf("2. sor - a kisebb mint b\n" );
		}
	else
		{
		printf("2. sor - a nem kisebb mint b\n" );
		}
	if ( a > b )
		{
		printf("3. sor - a nagyobb mint b\n" );
		}
	else
		{
		printf("3. sor - nem nagyobb, mint b\n" );
		}
	/* a es b erteke megvaltozik */
	a = 5;
	b = 20;
	if ( a <= b )
		{
		printf("4. sor - a kisebb vagy egyenlo  b\n" );
		}
	if ( b >= a )
```

```
            {
            printf("5. sor- b nagyobb vagy egyenlo a\n" );
            }

    return 0;
      }
```

4.3 Logikai operátorok

Operátor	Jelentés
&&	Logikai és operátor. Ha **mindkét** operandus nullától különböző, akkor igaz értéket ad
||	Logikai vagy operátor. Ha **bármelyik** operandus nullától különböző, akkor igaz értéket ad
!	Logikai negáció operátor. Ha a feltétel igaz, hamis értéket ad, ha a feltétel hamis, igaz értéket ad.

```
#include <stdio.h>

int main()
      {
      int num1 = 30;
      int num2 = 40;

      if(num1>=40 || num2>=40) /* hamis vagy igaz */
      printf("Vagy muvelet vegrehajtva\n");

      if(num1>=20 && num2>=20) /* igaz es igaz */
      printf("Es muvelet vegrehajtva\n");

      if( !(num1>=40)) /* hamis negalva */
      printf("Negalas vegrehajtva");
      getchar();
      return(0);
      }
```

4.4 Bit operátorok

Operátor	Jelentés
&	Bitenkénti és operátor. Ha **mindkét** operandus 1 különböző, akkor 1 lesz.
|	Bitenkénti vagy operátor. Ha **bármelyik** operandus 1, akkor 1 lesz.
^	Bitenkénti kizáró vagy. Ha csak az egyik operandus 1, akkor 1 lesz.

~	Bitenkénti negálás. 1 esetén 0, 0 esetén 1.
<<	Eltolás balra egy bittel
>>	Eltolás jobbra egy bittel

4.5 Értékadás operátorok

Operátor	Jelentés
=	Egyszerű értékadás. A jobb oldali operandus értékét felveszi a baloldali operandus
+=	Összeadás és értékadás. A bal oldali operandust megnöveli a jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
-=	Kivonás és értékadás. A bal oldali operandust csökkenti a jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
*=	Szorzás és értékadás. A bal oldali operandust megszorozza a jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
/=	Osztás és értékadás. A bal oldali operandust elosztja a jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
%=	Osztás és értékadás. A bal oldali operandus maradékát képzi a jobboldali operandus értékével való osztáskor. Az eredményt a bal oldali operandus tárolja
<<=	Biteltolás balra és értékadás. A bal oldali operandust eltolja a jobboldali operandus értékének megfelelő bittel. Az eredményt a bal oldali operandus tárolja
>>=	Biteltolás jobbra és értékadás. A bal oldali operandust eltolja a jobboldali operandus értékének megfelelő bittel. Az eredményt a bal oldali operandus tárolja
&=	Bitenkénti és és értékadás. A bal oldali operandus bitenkénti és műveletét képzi jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
^=	Bitenkénti kizáró vagy és értékadás. A bal oldali operandus bitenkénti kizáró vagy műveletét képzi jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja
|=	Bitenkénti vagy és értékadás. A bal oldali operandus bitenkénti vagy műveletét képzi jobboldali operandus értékével. Az eredményt a bal oldali operandus tárolja

4.6 Egyéb operátorok

Operátor	Jelentés
sizeof()	a kifejezés méretét adja meg bájtokban
&	Egy változó címét adja meg
*	pointert definiál egy változóra
?:	Feltétles operator, pl.: feltetel ? x_ertek : y_ertek Ha ’feltetel’ igaz, akkor x_ertek egyébként y_ertek lesz az eredménye
,	Összekapcsolja a kifejezéseket

Példa az & operátor használatára: a scanf a num változó **címét** kapja meg. A zárójeles kifejezés értékelődik ki először, képezzük num maradékát a 2-vel való osztáskor. Ha ez 0, akkor a `printf("Paros")` hajtódik végre. Ha nem, akkor a : utáni rész `printf("Paratlan")`.
Figyeljük még meg a moduló (%) operátort és a == (egyenlő) operátort!

```
#include<stdio.h>

int main()
      {
      int num;

      printf("Adj meg egy szamot : ");
      scanf("%d", &num);

      (num % 2 == 0) ? printf("Paros") : printf("Paratlan");
    return 0;
      }
```

Példa a sizeof() használatára:

```
#include <stdio.h>

int main()
      {
      int ivar = 100;
      char cvar = 'a';
      float fvar = 10.10f;
      double dvar = 10.20;

      printf("%d\t", sizeof(ivar));
```

```
printf("%d\t", sizeof(cvar));
printf("%d\t", sizeof(fvar));
printf("%d\n", sizeof(dvar));
/* A kiiratas 4   1     4     8 */
getchar();
return 0;
}
```

4.7 Operátorok kiértékelési sorrendje

Az Operátorok kiértékelési sorrendje, azaz a precedencia határozza meg azt, milyen sorrendben történik az összetett kifejezések kiértékelése. például az x = 6 + 5 * 4 az 26 lesz, nem pedig 44, mert a szorzás magasabb precedenciájú, mint az összeadás. Az alábbi táblázat a magasabb precedenciájú operátoroktól az alacsonyabb precedenciájúkig felsorolja az operátorokat, néhány –a C nyelvhez tartozó – operátorral kiegészítve a fentieket.

Kategória	Operátor	Kiértékelés
Postfix	() [] -> . ++ - -	Balról jobbra
Unáris	+ - ! ~ ++ - - (type)* & sizeof	Jobbról balra
Multiplikatív	* / %	Balról jobbra
Additív	+ -	Balról jobbra
Eltolás	<< >>	Balról jobbra
Összehasonlító	< <= > >=	Balról jobbra
Értékadó	== !=	Balról jobbra
Bitenkénti ÉS	&	Balról jobbra
Bitenkénti KIZÁRÓ VAGY	^	Balról jobbra
Bitenkénti VAGY	|	Balról jobbra
Logikai ÉS	&&	Balról jobbra
Logikai VAGY	||	Balról jobbra
Feltételes	?:	Jobbról balra

4.8 Balérték

A balérték fogalmát egy példán keresztül mutatjuk be. Vegyük a következő kódrészletet:

```
int n;
n = 5;
```

Az egyenlőségjel bal oldalán található a balérték, jobb oldalán a jobbérték. A balértékre vonatkozóan megkötések vannak. Vizsgáljuk meg az alábbi kódrészleten keresztül:

```
#define MAX 20
enum {HETFO,KEDD,SZERDA};
struct haromszog{
      int  a,b,c;
      };
int main()
      {
      int n;
      5 = n; /* hiba: balertek nem lehet konstans */
      MAX = 20;   /* hiba: balertek nem lehet makro */
      HETFO = 1; /* hiba: balertek nem lehet enum */
      haromszog = {3, 4, 5}; /* hiba: balertek nem lehet tipus */
      return(0);
      }
```

Eek mind hibát eredményeznek. A jobbérték fogalmát a következő példák mutatják be:

```
#define MAX 20

int main()
      {
      int n;
      int m = 7;
      n = 5; /* jobbertek konstans */
      n = 5 + 4; /* jobbertek kifejezes */
      n = MAX;    /* jobbertek makro */
      n = m;      /* jobbérték valtozo */
      return(0);
      }
```

5 Elágazások

Az elágazások segítségével befolyásolhatjuk, hogy egy kódblokk végrehajtásra kerüljön-e vagy sem.

5.1 Az if utasítás

Az if utasítás tartalmaz egy feltételt. Ha ez a feltétel igaz, akkor a kódblokk végrehajtódik. A C programozási nyelven azok a kifejezések vesznek fel igaz értéket, amelyeknek az eredménye nem nulla. Az if utasítás alapszintaktikája a következő:

```
if (feltetel)
    futtasd_le_ezt_a_kodot();
```

Az if utasítás után nem csak egy kódsort futtathatunk le, hanem egy egész kód blokkot is, ezt tegyük kapcsos zárójelbe. Jó programozási praktika, ha mindig kapcsos zárójeleket használunk

```
if (i > 0)
    {
    gyok = sqrt(i);
    printf("%d negyzetgyoke %f\n", i, gyok);
    }
```

Az else utasítás az if utasítással használatos. Ha a feltétel igaz, akkor az if utáni kódblokk fut le. Ha hamis, akkor az else ág kerül végrehajtásra.

```
if (feltetel)
    {
    /* ha a feltetel igaz, ez a kod fut le */
    }
else
    {
    /* ha a feltetel hamis, ez a kod fut le */
    }
```

Az előző példát kiegészítve, ha i pozitív, akkor az sqrt függvényhívás kiszámítja a négyzetgyökét, és kiírja azt. Egyébként kiírjuk a számot, és azt, hogy negatív

```
if (i > 0)
    {
    gyok = sqrt(i);
    printf("%d negyzetgyoke %f\n", i, gyok);
    }
else
    {
    printf("%d negativ", i);
    }
```

Az else if utasítás akkor hasznos, ha kettőnél több ágon futhat le a kódunk. Egészítsük ki a négyzetgyök számoló algoritmusunkat reciprok számlálással .

```
if (i > 0)
    {
    gyok = sqrt(i);
    reciprok = 1.0 / i;
```

```
    printf("%d negyzetgyoke %f reciproka %f\n", i, gyok, reciprok);
    }
else if (i != 0)
    {
    reciprok = 1.0 / i;
    printf("%d recoproka %f", i, reciprok);
    }
else
    {
    printf("a szam nulla");
    }
```

Tehát ha i pozitív, akkor az if ág hajtódik végre. Egyébként ha i nem nulla, akkor az else if csak a reciprokot számolja. Ha pedig i nulla, akkor az utolsó else ág ír a képernyőre. Tetszőleges számú else if ágat definiálhatunk.

5.2 A switch…case utasítás

Ha a kiértékelendő feltételek száma nagy, akkor ezt a kódot nagyszámú if..else if ág segítségével kell megoldanunk. Ez nem szerencsés. Ilyenkor használhatjuk a switch…case utasítást, amelynek a szintaktikája

```
switch (valtozo)
    {
    case ertek1:
      /*Ez a kod akkor fut le, ha valtozo == ertek1*/
      break;
    case ertek2:
      /*Ez a kod akkor fut le, ha valtozo == ertek2*/
      break;
    ...
    default:
      /*Ez a kod akkor fut le, ha valtozo nem egyenlo egyik ertekkel sem*/
      break;
    }
```

A default ág megadása nem kötelező, ide akkor jutunk el, ha a case ágak egyikével sem egyenlő a switch utasításban megadott változó.
Figyeljük meg, hogy az egyes case ágakat break; utasítás választja el. A vezérlés itt lép ki a case ágból

```
switch (valtozo)
    {
    case ertek1:
    case ertek2:
```

```
      /* Ez a kod akkor fut le, ha valtozo == ertek1
      vagy valtozo == ertek2*/
      break;

    case ertek3:
      /* Ez a kod akkor fut le, ha valtozo == ertek3 */
    case ertek4:
      /* Ez a kod akkor fut le, ha valtozo == ertek4 */
      break;
    }
```

Figyeljük meg, hogy a case ertek1: és case ertek2 ugyanazt a kódot futtatja le. Ha a valtozo==ertek3, akkor a kód a következő break; utasításig fut, azaz az ertek4-hez tartozó kód is lefut.

```
#include <stdio.h>

void filemenu()
{
    printf( "File menu megnyitasa" );
}
void editmenu()
{
    printf( "Szerkesztes menu megnyitasa" );
}

int main()
{
    int input;

    printf( "1. File\n" );
    printf( "2. Szerkeszt\n" );
    printf( "9. Kilep\n" );
    printf( "Valassz: " );
    scanf( "%d", &input );
    switch ( input ) {
        case 1:             /* ez itt kettospont, nem pontosvesszo */
            filemenu();
            break;
        case 2:
            editmenu();
            break;
        case 9:
            printf( "Viszlat\n" );
            break;
        default:
```

```
            printf( "Hibas bevitel!\n" );
            break;
    }
    getchar();
}
```

Ebben a példában egy egyszerű menüválasztó kódot írtunk meg. Bekérjük az input értékét. 1 esetén a filemenu, 2 eseten az editmenu függvényeket hívjuk meg. A 9 érték esetén a „Viszlat” üzenetet írjuk ki. Egyébként pedig a hibás bevitelre figyelmeztető szöveget látjuk.

6 Ciklusok

Ciklusoknak azokat a kódsorokat nevezzük, amelyek ismétlődnek. A programozás egyik legalapvetőbb feladata az, hogy végrehajtson kódblokkokat újra meg újra. Gondoljuk csak át, ennek milyen hatása van: csak egy aprócska kódblokkot írunk meg, de a ciklusban történő végrehajtás miatt a kód jelentősége nagymértékben megnövekszik.
Mielőtt folytatnánk a tanulást egy dolgot tisztázni kell: mit jelent a true (igaz) és a false (hamis) érték a C programozási nyelven. Minden nem nulla számot igaznak értékelünk ki, és hamisnak tekintjük azokat a kifejezéseket, amelyek zéró értéket vesznek fel.
Háromféle ciklusszervező utasítás van:

- for
- while
- do…while

Ezeket az alábbiakban ismertetjük.

6.1 A for ciklus

A for ciklus talán a leggyakrabban használt ciklus.

```
for (valtozo_inicializalas; feltetel; valtozo_modosítás)
      {
      /* Végrehajtandó kódblokk amíg a feltétel igaz*/
      }
```

A változó inicializálás részben deklarálhatunk egy változót és értéket adhatunk neki, vagy egy már létező változónak adhatunk értéket. A második részben megvizsgáljuk a feltételt. Ha a feltétel kiértékelésekor igaz értéket kapunk, akkor folytatódik a ciklus a következő iterációval. A változó módosítás részben kezelhetjük legegyszerűbb módon a ciklusváltozónkat. Tipikusnak tekinthető az i++, vagy az i = i + 10 alakú értékadás. Ha akarjuk, akkor itt egy

függvényhívást is elhelyezhetünk, amely semmit sem változtat a változóinkon, de valamilyen egyéb kódot végrehajt.
A for ciklusszervező utasításban pontosvesszővel kell elválasztani ezeket a részeket. Érdekesség, hogy a három rész közül bármelyiket üresen hagyhatjuk. A következő utasítás kiértékelésekor a feltétel igaz lesz és a kapcsos zárójelek közötti kódblokk addig ismétlődik, amíg valami meg nem állítja.

```
for (;;) { /* Folyamatosan ismetlodo kod */}
```

Példa

A feltétel ellenőrzése a következő iteráció megkezdése előtt történik. Azaz amikor i értéke 10 (azaz az i < 10 hamis lesz), a ciklusból kilépünk és a következő utasítással folytatódik a program végrehajtása.

```
#include <stdio.h>

int main()
{
    int i;
    /* a ciklus addig megy, amig i < 10*/
    for ( i = 0; i < 10; i++ ) {
     printf( "%d\n", i );
    }
    getchar();
}
```

A program egy olyan ciklust használ, ahol az i értéke nulla lesz, majd amíg az i értéke kisebb, mint 10, a printf függvénnyel kiírjuk i értékét. Ezután i értéke eggyel növekszik, és a feltétel teljesüléséig újabb iterációt hajtunk végre.

6.2 A while ciklus

Az utasitás szintaktikája egyszerű:

```
while (feltetel)
{
/* Vegrejajtando kod amig a feletel igaz*/
}
```

A tipikus feltétel az (i == 1) vagy (i != 1) alakú, de természetesen bármilyen boolean kifejezés lehet a zárójelben, amit a C nyelv elfogad. A while ciklusban nem maradhat üresen a feltétel.
Példa

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

```
int main()
{
  int i = 0;  /* Valtozo deklaralas */

  while ( i < 10 ) { /* amig i kisebb mint 10 */
      printf( "%d\n", i );
      i++;              /* Megnoveljuk i erteket */
  }
  getchar();
}
```

Ez a példa ugyanazt az eredményt adja, mint az előző.

6.3 A do…while ciklus

A do…while ciklus akkor alkalmazható jól, amikor a ciklus legalább egyszer le kell hogy fusson
A szintaktikája

```
do {
/* Folyamatosan ismetlodo kod */
} while (feltetel)
```

Ez az utasítás a feltételt a végén értékeli ki, azaz a kódblokk legalább egyszer biztosan lefut. Ha a feltétel igaz, akkor a blokk elejére visszaugrunk és újabb iterációt hajtunk végre.

Példa:

```
#include <stdio.h>

int main()
{
  int i;

  i = 0;
  do {
    /* "Hello, world!" egyszer kiirjuk pedig a feltetel hamis*/
      printf( "Hello, world!\n" );
  } while ( i != 0 );
  getchar();
}
```

Fontos szintaktikai szempontból, hogy a do…while után a pontosvesszőről ne feledkezzünk el.

Kicsit zavaró, hogy a sima while esetén pedig nem kell a pontosvessző. Az előző példa legfontosabb eleme, hogy a ciklus egyszer akkor is lefut, ha a feltétel már a kezdetekor hamis.

6.4 A break és continue utasítások

A ciklusok működésének két legfontosabb utasítása a break és continue. A break utasítás azonnal kilép a ciklusból, függetlenül attól, hogy a feltétel igaz vagy hamis értéket vesz fel. Akkor szokás alkalmazni, amikor a ciklusból valami egyedi, speciális esetben ki akarunk lépni.
Emlékszünk még az „üres” for ciklusunkra ?
Abból például a break utasítással léphetünk ki.

```
while(true)
{
megfigyeljuk_a_madarakat(); /* a fuggvenyt folyamatosan meghivjuk */
if (madarak_elrepultek)     /* kilepunk */
    {
    break;
    {
}
```

Bevett implementációs mechanizmus, hogy egy végtelen ciklust hozunk létre, amiből bizonyos esetekben a break utasítással lépünk ki.
A másik utasítás, amely befolyásolja a ciklus lefutását az a continue. Ha a ciklus kódjának futtatása eléri a continue utasítást, akkor az éppen futó iteráció folytatása helyett a rákövetkező iterációs lépés végrehajtásával folytatjuk.

```
int main()
{
for (int i = 1; i < 100; i++)
    {
    if ((primszam(i) == true))
        {
        continue;
        }
    faktorizal(i);
    }
}
```

Tételezzük fel, hogy van két függvényünk. A faktorizal() függvény egy egész számot felbont osztóira. Ez egy időigényes művelet, ezért prímszámokra nem akarjuk meghívni. A primszam() függvény pedig igaz értéket ad vissza, ha a kapott egész szám prím. A ciklusunkat úgy írtuk meg, hogy az egytől 100-ig az egész számokra hajtsa végre a faktorizációt. Prímszámok esetén pedig folytassa a következő iterációval a kód végrehajtását.

7 Egydimenziós tömbök és mutatók

7.1 Elméleti alapok

A tömb összetett adattípus. Olyan objektumok halmaza, amelyek azonos típusúak és a memóriában egymás után helyezkednek el. Deklaráláskor meg kell adni, hogy milyen típusú adatokból hány darabot szeretnénk tárolni a tömbben: *tipus tombnev[meret]*. A C nyelvben csak fordításkor ismert, fix méretű tömb deklarálható. A tömbelemek sorszámozása 0-tól méret-1-ig tart, de a fordító nem ellenőrzi az indexelés helyességét. A tömböt kezelni elemenként tudjuk: a tömbelemek önálló változók. A C nyelvben semmilyen művelet, szabványos függvény-könyvtár nincs a tömbök kezelésének támogatásához.
A tömb neve a tömb által elfoglalt memóriaterület kezdőcíme, azaz egy mutató. Olyan mutató, amelynek értéke nem változtatható meg; tehát nem lehet balérték. Mivel az egydimenziós tömbök (vektorok) és az egyszeres indirektségű mutatók között 100%-os a tartalmi és formai analógia, ezért minden művelet, ami tömbindexeléssel elvégezhető, mutatók segítségével is megvalósítható, és fordítva.

7.2 Gyakorló feladatok

7.2.1 Feladat:
Olvassunk be 5 egész számot egy tömbbe és határozzuk meg a tömb elemeinek átlagtól való eltérését. Oldjuk meg a feladatot mutató használatával is.
Az átlag kiszámításához szükségünk van egy 0-val inicializált változóra, amelyhez minden iterációban hozzáadjuk az aktuálisan feldolgozás alatt álló tömbelem értékét. A ciklus végén az összeget elosztjuk az elemek számával, hogy megkapjuk az átlagot. Az átlagtól való eltérések meghatározásához ismételten ciklusban kell feldolgozni a tömbelemeket: minden iterációs lépésben a ciklusváltozóval megadott indexű tömbelem értékéből kivonjuk az átlagot. Így megkapjuk az adott tömbelem átlagtól való relatív eltérését.

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

int main() {
    int szam[N], i, *p;
    double atlag = 0.0;
    for (i=0; i<N; i++) {                  /* elemek beolvasása */
        printf("%d. elem: ", i+1);
        scanf("%d", &szam[i]);
        atlag += szam[i];                  /* elemek összegzése */
    }
```

```
    atlag /= N;                             /* átlag kiszámítása */
    printf("\nAz atlag: %.2f\n", atlag);
    for (i=0; i<N; i++) /* tömbelemek és átlagtól való eltérés kiírása */
        printf("%d. \t %d \t %.2f\n", i, szam[i], szam[i]-atlag);

    /* mutató használatával */
    // a p pointer mindvégig a tömb első elemére mutat; i a ciklusváltozó
    for (p=szam, i=0; i<N; i++)
        printf("%d. \t %d \t %.2f\n", i, *(p+i), p[i]-atlag);
    // a p pointer mindig a tömb feldolgozás alatt álló elemére mutat;
    // az i sorszám csak a kiíratáshoz kell
    for (p=szam, i=0; p<=&szam[N-1]; i++, p++)
        printf("%d. \t %d \t %.2f\n", i, *p, (*p)-atlag);
    return 0;
}
```

7.2.2 Feladat:

Inicializáljunk egy 10 elemű *integer* tömböt. Valósítsuk meg rajta a lineáris keresés algoritmusát. Szükség van egy találat jelző változóra, ami kezdetben 0 (azaz 'nem talált') értékű. Ciklusban feldolgozva a tömb elemeit, minden iterációban összevetjük a keresett értéket a ciklusváltozóval adott indexű tömbelem értékével. Ha megegyeznek, megtaláltuk a keresett értéket: a találat jelző változó értékét 1-re, azaz igazra állítjuk és leáll az algoritmus. Amíg nincs meg a keresett érték, illetve van még feldolgozatlan tömbelem, léptetjük a ciklusváltozót és vesszük a következő tömbelemet.

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

int main() {
    int tomb[N] = { 6, 12, -3, 7, 4, 9, 56, 34, 0, -21 };
    int keresett, i=0, talalt=0;
    printf("\nA keresett szam: ");
    if (scanf("%d", &keresett)!=1) {    /*ellenőrzött adatbeolvasás*/
        printf("\nHibas adat!");
        exit(-1);                        /*hiba esetén leáll a program*/
    }
    else {
        while (i<N && !talalt) { /*amíg van elem és nem találtuk még meg*/
            if (tomb[i] == keresett) talalt=1;
            i++;
        }
    }
    if (talalt) printf("\nA keresett szam a tomb %d. eleme.\n", i);
    else printf("\nA keresett szam nem eleme a tombnek.\n");
```

```
    return 0;
}
```

7.2.3 Feladat:

Véletlenszámokkal töltsünk fel egy 10 elemű tömböt és állapítsuk meg a sorozat szélsőértékeit. A véletlenszámok előállításához könyvtári függvényeket használunk. Az *srand()* függvény inicializálja a véletlenszám generátort, majd az alábbi képlettel előáll egy, a *[alsóhatár, felsőhatár)* intervallumba eső szám: *rand()%(felsőhatár-alsóhatár)+alsóhatár*. Figyelem! Ha felülről zárt intervallumra van szükségünk, *felsőhatár+1*-el számoljunk.

```
#include <stdio.h>
#include <stdlib.h>           //srand(), rand() függvények hívásához
#include <time.h>             //time() függvény hívásához
#define N 10

int main() {
    int i, j, felsoh=101, alsoh=1, tomb[N], min, max;
    srand(time(0));           //véletlenszám generátor inicializálása
    for (i=0; i<N; i++)      //1 és 100 közötti számokkal feltölti a tömböt
        tomb[i]=rand()%(felsoh-alsoh)+alsoh;
    printf("A tomb elemei: \n");
    for (i=0; i<N; i++)
        printf("%d, ", tomb[i]);
    min=tomb[0];
    for(i=0; i<N; i++)
        if(tomb[i]<min) min=tomb[i];
    printf("\nA sorozat legkisebb eleme: %d", min);
    max=tomb[0];
    for(i=0; i<N; i++)
        if(tomb[i]>max) max=tomb[i];
    printf("\nA sorozat legnagyobb eleme: %d", max);
    printf("\nA sorozat kozepso eleme: %d", N%2?tomb[N/2]:tomb[N/2-1]);
    return 0;
}
```

7.2.4 Feladat:

Inicializáljunk egy 10 elemű valós tömböt és az elemeit írjuk ki fordított sorrendben, illetve összekeverve. A tömb összekeverése a tömbelemek véletlenszerű felcserélését jelenti. Mivel a tömbindexek alsó értéke 0, ezért a véletlen értéket előállító képlet egyszerűbben felírható.

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

```
int main() {
    int i, j;
    double elem;
    double tomb[] = {6.23,12.5,-3.8,7.75,4.9,9.5,56.0,34.2,0.25,-21.4};
    int meret = sizeof(tomb)/sizeof(double);    //tömb mérete
    printf("\nTombelemek forditott sorrendben: \n");
    for (i=(meret-1); i>=0; i--)                //tömb fordított bejárása
        printf("%.2f, ", tomb[i]);
    printf("\nTombelemek ossze-vissza: \n");
    srand(time(0));
    for(i=0; i<meret; i++) {
        j = rand()%meret;       //véletlenszerűen ad egy tömbindexet
        elem = tomb[j];
        tomb[j] = tomb[i];
        tomb[i] = elem;
    }
    for (i=0; i<meret; i++)
        printf("%.2f, ", tomb[i]);
    return 0;
}
```

7.2.5 Feladat:

Írjunk C programot az 5-ös lottó számainak véletlenszerű előállítására. Azért kell tömböt használjunk a feladathoz, mert 5 különböző számra van szükségünk. Az ismétlődés elkerülése érdekében eltároljuk az érvényes számokat és a következő jelölt előállításakor ezeket figyelembe vesszük.

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

/* 1 és 90 közötti véletlenszám előállítása; előfordulhat ismétlődés
int main() {
    srand(time(0));
    int i, N=5;
    for (i=1; i<=N; i++) {
        printf("%d. szam: %d\n", i, rand()%90+1);
    }
    return 0;
}*/

/* 1 és 90 közötti véletlenszám előállítása ismétlődés nélkül */
int main() {
    srand(time(0));
```

```
    int i, j, jelolt, N=5, talalt=0;
    int lotto[5];
    lotto[0] = rand()%90+1;
    for (i=1; i<N; i++) {
        jelolt = rand()%95+1;           //véletlenszám előállítása
        for (j=0; j<i; j++) {
            if(lotto[j]==jelolt) {      //ha már volt, elvetjük
                talalt=1;
                break;
            }
        }
        if(talalt==0) lotto[i]=jelolt;  //ha még nem volt, eltároljuk
    }
    for(i=0; i<N; i++) {                //kiírás
        printf("%d. szam: %d\n", i+1, lotto[i]);
    }
    return 0;
}
```

Vegyük észre, hogy nem lehet kikerülni a tömb használatát, ha egy elvégzendő művelethez szükség van az összes elemre, vagy azokat nem sorrendben kell feldolgozni.

8 Függvények

8.1 Elméleti alapok

A függvény olyan névvel ellátott programegység, amely a program más részeiből annyiszor hívható, ahányszor szükség van a függvényben definiált tevékenység sorozatra. A strukturált programozási elvet követve a C program több, jól meghatározott feladatot végrehajtó függvényből áll. Saját függvényeinket a következő szintaxissal definiáljuk a programon belül bárhol, függvényen kívül (a < > között megadott részek opcionálisak).

```
visszatérési_típus függvény_neve ( <paraméter deklarációs lista> )      //fejléc
{
        <lokális deklarációk és definíciók>
        utasítások                                                      függvénytörzs
}
```

A függvény visszatérési típusa a függvény által előállított érték típusa. Ez tetszőleges lehet, kivéve tömb. A visszaadott érték a függvénytörzs return utasításában szereplő kifejezés értéke.

Az a függvény, amely nem ad vissza értéket (return ;) void visszatérési típusú. A függvény lokális változói a függvény meghívásakor jönnek létre, csak a függvényen belül láthatók és a függvény futásának végén törlődnek a memóriából.
A függvény paraméterlistája a várt paraméterek típusát és nevét sorolja fel. Ez lehet üres (void). Függvényhíváskor az argumentumlista tartalmazza az aktuális paraméterértékeket, amelyek átmásolásra kerülnek a paraméterlista megfelelő tagjába. Ezért a fordító a paraméterlista és az argumentumlista között típusegyeztetést végez: számuk és típusuk rendre meg kell egyezzen. Ahhoz, hogy egy függvény hívható legyen, a hívás helye előtt (függvényen kívül) el kell helyezni a függvény deklarációját vagy prototípusát, ami a függvény fejléce pontosvesszővel lezárva. Ez a magyarázata annak, hogy az adott forrásfájlon kívül definiált függvények (pl. könyvtári függvények) is csak akkor hívhatók, ha a modul elején beépítjük a deklarációjukat tartalmazó header állományt (#include <stdio.h> vagy saját függvénykönyvtár esetén #include “sajat.h”).

8.2 Gyakorló feladatok

8.2.1 Visszatérési értékkel nem rendelkező függvények

8.2.1.1 Feladat:
Rajzoljunk *-okból adott méretű 5-öst a képernyőre. A rajzoló függvénynek nincs bemeneti paramétere (üres v. void a paraméterlista) és nincs visszatérési értéke. Eredménye a szabványos kimenetre kerül.

```
#include <stdio.h>

void otos_rajzol(void);      //függvény deklaráció

int main() {
    int oldal;
    printf("A sikeres futas jutalma: \n\n");
    otos_rajzol();           //függvényhívás
    return 0;
}

/* Ötösrajzoló függvény definíciója */
void otos_rajzol(void) {                  //függvény fejléce
    int i, sor, oldal=4;                  //az ötös szárának hossza
    int magassag=(2*oldal)-1;             //biztosan páratlan szám
    int felezo=(magassag/2)+1;            //egész osztás!
    for (sor=1; sor<=magassag; sor++) {
        if (sor==1 || sor==felezo || sor==magassag ) {  //teljes sorok
```

```
            for (i=1; i<=oldal; i++) printf(" * ");
        }
        else {
            if (sor<felezo) {
                printf(" * ");
                for(i=1; i<=oldal-1; i++) printf("   ");
            }
            else {
                for(i=1; i<=oldal-1; i++) printf("   ");
                printf(" * ");
            }
        }
        printf("\n");                //sor vége
    }
    return ;                         //nincs visszatérési érték
}
```

8.2.1.2 Feladat:

Számelméleti algoritmusoknál gyakori feladat, hogy egy bizonyos intervallumba eső számokról kell megállapítani, hogy rendelkeznek-e valamely adott tulajdonsággal. Az intervallum alsó és felső határának beolvasása jól körülhatárolható feladat: a hívó függvényben deklarált két változó megkapja a felhasználó által megadott (szabványos bemenetről beolvasott) értékeket. A hívott függvény a híváskor megadott argumentumok értékének másolatát kapja meg bemeneti paraméterként. Ha híváskor a deklarált (de általában inicializálatlan) változók értékét adnánk át, két hibát is elkövetnénk: 1) inicializálatlan változó értékre nem szabad hivatkozni; 2) az érték szerint átadott paraméterek a hívott függvényben lokális változóként viselkednek, azaz a függvény futása után megszűnnek. Ezért a változóknak nem az értékét, hanem a címét adjuk át. A függvény hatása sikeres futás esetén az, hogy a két változó értéket kap. Amennyiben a függvény elvégzi a beolvasott érték ellenőrzését (nem a hívó függvény feladata), akkor nincs visszatérési értéke.

```
#include <stdio.h>

int szamvizsgalat(int szam) { return 0; }    // ezt most nem definiáljuk

void beolvas(int *hatar) {          // adatbeolvasó függvény definíciója
      if (scanf("%d", hatar)!=1) {
        printf("Hibas bemenet!");
        exit(-1);
      }
      return ;
}
```

```
int main( ) {
      int alsohatar, felsohatar, i;
      printf("Adja meg az intervallum also hatarat: ");
      beolvas(&alsohatar);
      do {                          // intervallum határok ellenőrzése
        printf("Adja meg az intervallum felso hatarat: ");
            beolvas(&felsohatar);
      } while (felsohatar < alsohatar);
      for (i=alsohatar; i<=felsohatar; i++){
            if (szamvizsgalat(i)==1)
                  printf("%d", i);
      }
      return 0;
}
```

8.2.2 Visszatérési értékkel rendelkező függvények

8.2.2.1 Feladat:

Az előző feladatot folytatva, keressük meg egy adott intervallumban a tükörszámokat. A vizsgálatot végző függvény paraméterként kap egy számot és 1-el tér vissza, ha a vizsgált számra teljesül a feltétel (azaz tükörszám), ill. 0-val ha nem. Tükörszámnak nevezzük azt a számot, amelyik megegyezik a fordítottjával. Tehát szükségünk lesz egy olyan függvényre is, ami előállítja a vizsgált szám fordítottját.

Adjuk meg a probléma rekurzív megoldását is. *Rekurzívnak nevezzük azt a függvényt, amely közvetlenül vagy közvetve önmagát hívja. Minden cikus megvalósítható rekurzióval és minden rekurzió megvalósítható ciklussal.*

Iteratív függvény:
- Amíg feltétel igaz
  - Ciklusmag
- Ciklus vége

Rekurzív függvény:
- Ha feltétel igaz
  - Ciklusmag
  - Rekurzív fv hívás
- Elágazás vége

```
#include <stdio.h>

int szamvizsgalat(int szam);
int szamfordit(int n);
int szamfordit_rek(int n, int ford);

void beolvas(int *hatar) {          // adatbeolvasó függvény definíciója
      if (scanf("%d", hatar)!=1) {
        printf("Hibas bemenet!");
```

```
        exit(-1);
    }
      return ;
}

int main( ) {
      int alsohatar, felsohatar, i;
      printf("Adja meg az intervallum also hatarat: ");
      beolvas(&alsohatar);
      do {                              // intervallum határok ellenőrzése
        printf("Adja meg az intervallum felso hatarat: ");
            beolvas(&felsohatar);
      } while (felsohatar < alsohatar);
      for (i=alsohatar; i<=felsohatar; i++){
            if (szamvizsgalat(i)==1)
                  printf("%d ", i);
      }
      return 0;
}

int szamvizsgalat(int szam) {
    if (szam==szamfordit(szam)) return 1;
    //if (szam==szamfordit_rek(szam,0)) return 1;
    else return 0;
}

int szamfordit(int n) {                   /* iteratív megoldás */
      int ford=0;
      while (n>0) {
            ford = ford*10+n%10;
            n = n/10;
      }
      return ford;
}

int szamfordit_rek(int n, int ford){      /* rekurzív megoldás */
    if (n>0)
       return szamfordit_rek(n/10, ford*10+n%10);
    else
       return ford;
}
```

8.2.3 A main függvény

8.2.3.1 Feladat:

Az 5-ös rajzoló program main függvényét írjuk át úgy, hogy bekérjük a felhasználótól az 5-ös

méretét, ami 3-nál nagyobb egész szám kell legyen. A main függvény visszatérési értéke egy egész szám, ami a program futás sikerességét jelzi. Sikeres futás esetén ez 0. A hibák jelzésére tetszőlegesen használhatunk egész számokat.

```
#include <stdio.h>

void otos_rajzol(int oldal) { ... }       //lásd 8.2.1.1 feladat

int main() {
    int oldal;
    printf("Adja meg az oldal hosszusagot (min.3): ");
    if(scanf("%d", &oldal)!=1) {        //nem egy egész számot adott meg
        printf("Hibas bemenet!");
        return 1;
    }
    if(oldal<3) {
        printf("Hibas bemenet!");        //3-nál kisebb számot adott meg
        return 2;
    }
    otos_rajzol(oldal);
    return 0;                            //sikeres programfutás
}
```

8.2.3.2 Feladat:

A program belépési pontja a main függvény, amit a program elindításakor az operációs rendszer hív meg. Két bemeneti paramétere van: a program indításakor megadott parancssori argumentumok száma és az argumentumokat tároló sztringtömb. Egy argumentum mindig van: a program neve. CodeBlocks fejlesztőkörnyezetben a parancssori argumentumok megadása: *Project – Set program's arguments*. Írjuk meg a main függvénynek átadott argumentumokat kiíró kódot.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
      int k;
      printf("Az atadott argumentumok szama: %d\n", argc);
      printf("Az atadott argumentumok listaja: \n");
      for (k=0; k<argc; k++)
            printf("%d. arg: %s\n", k, argv[k]);
      return 0;
}
```

9 Alapalgoritmusok

9.1 Számlálás, összegzés

9.1.1 Feladat:

Számítsuk ki adott N-ig a prímszámok számtani és mértani átlagát. A számtani átlag a sorozat elemeinek összege elosztva a sorozat elemszámával. Tehát a számtani átlag számításához most két programozási alaptételt is fel kell használnunk: összegezni kell 1-től N-ig a prímszámokat és meg kell határozni a darabszámukat. A mértani átlag a sorozat elemeinek szorzata az elemszámmal megegyező gyök alatt. Ehhez a számlálás alaptételt és az összegzés szorzatképzéshez módosított változatát fogjuk alkalmazni.

```
#include <stdio.h>
#include <math.h>
/* saját függvények deklarációja */
double szatl_szamit(int meddig);
double matl_szamit(int meddig);
int primszam_e(int szam);

int main() {
    int n;
    printf("Meddig szamolunk? ");
    scanf("%d", &n);
    double szatlag = szatl_szamit(n);
    double matlag = matl_szamit(n);
    printf("\n%d-ig a primszamok szamtani atlaga: %.2f", n, szatlag);
    printf("\n%d-ig a primszamok mertani atlaga: %.2f", n, matlag);
    return 0;
}

double szatl_szamit(int meddig) {         //számtani átlag számítása
    int i, db=0;              //számláló 0-ról indul
    double osszeg=0;          //összeg 0-ról indul
    for(i=1; i<=meddig; i++)
        if(primszam_e(i)) {
            osszeg += i;      //az összeghez hozzáadjuk a vizsgált számot
            db++;             //számlálót növeljük
        }
    return osszeg/db;
}

double matl_szamit(int meddig) {          //mértani átlag számítása
    int i, db=0;              //számláló 0-ról indul
    double szorzat=1;         //szorzat 1-ről indul
```

```
    for(i=1; i<=meddig; i++)
        if(primszam_e(i)) {
            szorzat *= i;   //a szorzathoz hozzávesszük a vizsgált számot
            db++;           //számlálót növeljük
        }
    return pow(szorzat,1/(double)db);   //math.h hatványozó függvénye
}

int primszam_e(int szam) {  //a szám prím, ha valódi osztóinak száma 0
    int oszto=2;
    while (oszto<=szam/2) {
        if(szam%oszto==0) return 0;     //nem prím
        oszto++;
    }
    return 1;                           //prím
}
```

9.2 Eldöntés, kiválasztás

9.2.1 Feladat:

Legyen adott a Miskolci Egyetem mérnök informatika alapképzésére 2015-ben jelentkezők (300 fő) pontjainak listája és a felvehető hallgatók létszáma (100 fő). Döntsük el, hogy egy adott pontszámmal felvételt nyerünk-e. Rendezetlen sorozat esetén végig kell nézni az összes elemet a kérdés megválaszolásához: meg kell számolni hány jelentkezőnek van magasabb pontszáma a megadottnál. Rendezett sorozat esetén csak a lista 100. elemét kell vizsgálni, hogy kisebb-e mint a megadott pontszám: ebben az esetben felvételt nyer a jelentkező.

```
#include <stdio.h>
#include <stdlib.h>
#define MAXPONT  400
#define MINPONT  180
#define LETSZAM  100                //100 fő a felvehető létszám

void lista_feltoltes(int *tomb, int meret) {
    int i;
    srand(time(0));
    for (i=0; i<meret; i++)
        tomb[i]=rand()%(MAXPONT+1-MINPONT)+MINPONT; //pontszámok: 180-400
    return ;
}

int dontes(int *tomb, int meret, int pontszam) {
    int i, db=0;
    for(i=0; i<meret; i++){
```

```
        if(tomb[i]>=pontszam) db++;    //hányan annak előtted?
    }
    if(db<LETSZAM) return 1;          //100 fő a felvehető létszám
    else return 0;
}

int main() {
    int N = 300;
    int tomb[N];
    int pont;
    lista_feltoltes(tomb, N);
    printf("Mennyi a pontszamod? ");
    scanf("%d", &pont);
    printf("%s",
      dontes(tomb, N, pont) ? "\nFelvetelt nyertel!" : "\nNem sikerult.");
    return 0;
}
```

9.2.2 Feladat:

Legyen adott az Alpok 100 legmagasabb csúcsának listája. Keressük meg a 3000 méteres csúcsok közül a legalacsonyabbat. A globális minimum keresés algoritmusát nem alkalmazhatjuk közvetlenül. Először meg kell keresni a sorozatban az első 3000 méteres csúcsot. Feltesszük, hogy ez a legkisebb és ehhez hasonlítva a többi 3000-nél nagyobb értéket meghatározzuk közülük a minimumot. A feladat másik megoldása, hogy az eredeti sorozatból először kiválogatjuk a 3000-nél nagyobb értékeket és erre az új sorozatra alkalmazzuk a minimum kereső eljárást.

```
#include <stdio.h>
#include <stdlib.h>
void lista_feltoltes(int *tomb, int meret);
int legkisebb_3ezres(int *tomb, int meret, int hatar);
int kivalogatas(int *tomb3000, int *tomb, int meret, int hatar);
int legkisebb(int *tomb3000, int ujmeret);

int main() {
    int meret=200, tomb[meret];
    lista_feltoltes(tomb, meret);
    printf("A legkisebb 3000 meteres csucs: %d m.",
                legkisebb_3ezres(tomb, meret, 3000));
    /* kiválogatással */
    int ujtomb[meret];
    int ujmeret = kivalogatas(ujtomb, tomb, meret, 3000);
    printf("\nA legkisebb 3000 meteres csucs: %d m.",
                legkisebb(ujtomb, ujmeret));
```

```
    return 0;
}

void lista_feltoltes(int *tomb, int meret) { //véletlenszámokkal
    int i;
    srand(time(0));
    for (i=0; i<meret; i++)
        tomb[i]=rand()%(4811-2000)+2000;  //2000 és 4810 közötti számok
    return ;
}

int legkisebb_3ezres(int *tomb, int meret, int hatar) {
    int i, min;
    for (i=0; i<meret; i++)            //első 3000-res csúcs keresése
        if(tomb[i] >= hatar) min = tomb[i];
    for (i=0; i<meret; i++)            //minimum keresés algoritmusa
        if (tomb[i] >= hatar && tomb[i] < min) min = tomb[i];
    return min;
}

int kivalogatas(int *tomb3000, int *tomb, int meret, int hatar) {
    int i, db=0;
    for(i=0; i<meret; i++)
        if(tomb[i] >= hatar) {
            tomb3000[db] = tomb[i];
            db++;
        }
    return db;
}

int legkisebb(int *tomb3000, int ujmeret) { //minimum keresés algoritmusa
    int i, min = tomb3000[0];
    for (i=0; i<ujmeret; i++) {
        if (tomb3000[i] < min) min = tomb3000[i];
    }
    return min;
}
```

9.3 Keresés

9.3.1 Feladat:

Adott a Forma-1 versenyzők 2014-es tabellája. Keressünk meg a ponttáblázatba egy megadott pontszámot. Az eljárás tulajdonképpen az eldöntés és a kiválasztás algoritmusainak együttes alkalmazása. Mivel a bemenő számsorozat csökkenő sorrendben rendezett, a lineáris és a

logaritmikus keresési tétel alkalmazásával is megoldható a feladat. Vegyük észre, hogy ha kihasználjuk a lista rendezettségét és ismerjük a sorozat adatainak eloszlását, a keresés irányának megfelelő megválasztásával (i=0…méret-1, vagy fordítva) befolyásolhatjuk a lineáris keresés gyorsaságát. Tekintsük ezt az esetet és a megadott pontszám legyen 100. Ekkor a lineáris kereső algoritmus 9 iteráció után áll le azzal az eredménnyel, hogy nincs a listán az adott pontszám, míg a logaritmikus kereső eljárás 4 iteráció után jut ugyanerre az eredményre. A standard bsearch() bináris kereső függvényt most nem tudjuk alkalmazni, mert az nem a keresett elem indexével tér vissza, hanem a keresett elemre mutató pointerrel (ami sikertelen keresés esetén NULL).

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

int lin_keres(int *tomb, int meret, int k);
int log_keres(int *tomb, int meret, int k);

int main() {
    int meret = 24;
    int tabella[24] = {384, 317, 238, 186, 167, 161, 134, 126, 96, 59, 55,
                       55, 22, 8, 8, 2, 2, 0, 0, 0, 0, 0, 0, 0};
    int pont, eredmeny;
    printf("Adj meg egy pontszamot: ");
    if(scanf("%d", &pont) != 1) {         /* ellenőrzött beolvasás */
        printf("Hibas adat!");
        exit(-1);
    }
    /* lineáris kereséssel */
    eredmeny = lin_keres(tabella, meret, pont);
    if ( eredmeny != 0)
      printf("%d ponttal %d. helyen all a versenyzo.\n", pont, eredmeny);
    else
        printf("%d pontszammal nincs versenyzo a listan.\n", pont);
    /* logaritmikus kereséssel */
    eredmeny = log_keres(tabella, meret, pont);
    if ( eredmeny != 0)
        printf("%d ponttal %d. helyen all a versenyzo.\n", pont, eredmeny);
    else
        printf("%d pontszammal nincs versenyzo a listan.\n", pont);

    return 0;
}
```

```
/* lineáris keresés */
int lin_keres(int *tomb, int meret, int k) {
    int i=0, talalt=0, hol=0;
    // while(i<meret && !talalt && k>=tomb[i]) - csökkenően rend. tömb
    // while (i<meret && !talalt && k<=tomb[i]) - növekvően rend. tömb
    while (i<meret && !talalt) {        // általános esetben
        if (tomb[i] == k) {
            talalt=1;
            hol=i+1;                    // tömbindex + 1
        }
        i++;
    }
    return hol;
}

/* logaritmikus keresés rendezett tömbben */
int log_keres(int *tomb, int meret, int k) {
    int talalt=0, hol=0;
    int also=0, felso=meret-1, kozepso;
    while (also <= felso && !talalt) {
        kozepso = (also+felso)/2;
        if (tomb[kozepso] == k) {
            talalt = 1;
            hol = kozepso+1;            // tömbindex + 1
        }
        /* csökkenő rendezettség esetén */
        if (tomb[kozepso] < k) felso = kozepso - 1;
        if (tomb[kozepso] > k) also = kozepso + 1;
    }
    return hol;
}
```

9.4 Rendezés

9.4.1 Feladat:

Programozás alapjai tárgyból adott a zh pontszámok sorozata. Készítsünk zh statisztikát! Első feladatként (a) rendezzük a listát saját eljárással és határozzuk meg hány sikeres dolgozat született (min. 25 pont). Második feladat (b) az stdlib.h standard függvényeit használva megállapítani, hogy van-e maximális pontszámú (50 pont) dolgozat.

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

```c
void feltolt(int *tomb, int meret);
void kiir(int tomb[], int meret);
int keres(int *tomb, int meret, int k);
void rendez(int *tomb, int meret);
void a_feladat(int * tomb, int N, int minpont);
void b_feladat(int * tomb, int N, int maxpont);

// int elemeket összehasonlító fv standard kereső és rendező eljárásokhoz
int cmpfunc(const void * a, const void * b){
      return ( *(int*)a - *(int*)b );
}

int main() {
    int N = 20, minpont = 25, maxpont = 50;
    int tomb[N];
    feltolt(tomb, N);
    a_feladat(tomb, N, minpont);
    b_feladat(tomb, N, maxpont);
    return 0;
}

void feltolt(int *tomb, int meret) {
    int i;
    srand(time(0));
    for (i=0; i<meret; i++)
        tomb[i]=rand()%50;     //0 és 50 közötti számok
    return ;
}

void kiir(int tomb[], int meret) {
    int i;
    for(i=0; i<meret; i++) {
        printf("%d. szam: %d\n", i+1, tomb[i]);
    }
    return ;
}

/* linearis keresés */
int keres(int *tomb, int meret, int k) {
    int i=0, talalt=0, hol=0;
    while (i<meret && !talalt) {
        if (tomb[i] >= k) { // az első 25 v. 25-nél nagyobb pontszám
            talalt=1;
            hol=i;
        }
```

```
        i++;
    }
    return hol;
}

/* minimumkiválasztásos rendezés - növekvő sorozat */
void rendez(int *tomb, int meret) {
    int i,j,minindex, temp;
    for(i=0; i<meret-1; i++){          // i: hova
        minindex=i;
        for(j=i+1; j<meret; j++)
            if( tomb[j]<tomb[minindex] ) minindex=j;
            if(minindex!=i){
                temp=tomb[minindex];  // csere
                tomb[minindex]=tomb[i];
                tomb[i]=temp;
            }
    }
    return ;
}

void a_feladat(int * tomb, int N, int minpont) {
    rendez(tomb, N);                    //növekvő sorrendbe rendez
    kiir(tomb, N);
    int minindex=keres(tomb, N, minpont);  //az első sikeres dolgozat
    printf("%d dolgozat sikeres.\n%d dolgozat sikertelen.\n",
            N-minindex, minindex);
    return ;
}

/* tömb rendezése és adott pontszám keresése standard függvényekkel */
void b_feladat(int * tomb, int N, int maxpont) {
    // stdlib.h gyorsrendező eljárása, növekvő sorrendbe rendez
    qsort(tomb, N, sizeof(int), cmpfunc);
    // stdlib.h bináris kereső eljárása
    int *poz = (int*) bsearch (&maxpont, tomb, N, sizeof(int), cmpfunc);
    if( poz != NULL )
        printf("Van maxpontos dolgozat!\n");
    else printf("Nincs maxpontos dolgozat.\n");
    return ;
}
```

10 Sztringek kezelése

10.1 Elméleti alapok

A C nyelvben nincs önálló sztring típus. A karaktersorozatokat egydimenziós karaktertömbben tároljuk, ahol a tömb utolsó eleme a '\0' lezáró karakter. A tömbös tárolás miatt nincsenek sztring műveletek sem definiálva. Kezelésük függvényeken keresztül történik. Ha sztringek tömbjére van szükségünk, azt kétdimenziós tömb vagy az optimálisabb helykihasználás miatt mutatótömb alakban adhatjuk meg. Kétdimenziós tömbös tárolás esetén az első dimenzió a tárolandó sztringek számát, míg a második dimenzió a sztringek számára egyenként lefoglalt memóriaterület méretét adja meg. Mutatótömbös tárolás esetén az egyes sztringekre mutató pointereket tároljuk egydimenziós tömbben. Mindkét esetben hivatkozáskor az első index egy adott sztringet, míg a második index a sztring egy adott karakterét határozza meg.

10.2 Gyakorló feladatok

10.2.1 Karaktertömb inicializálása, elemenkénti kezelése

10.2.1.1 Feladat:

Felhasználó által megadott sztringet írjunk ki virágnyelven: minden magánhangzó után tegyük hozzá a 'v+magánhangzó' karaktersorozatot.

```
#include <stdio.h>
#include <ctype.h>            //tolower() karakterkonverziós függvényhez
void viragnyelven_kiir(char sztring[]);
int main() {
    /* inicializálás sztring-konstanssal:
                  a lezáró karaktert a fordító hozzáfűzi */
    // char szoveg[100] = "Varga Erika";
    char szoveg[100];
    printf("Adj meg egy sztringet (max. 100 karakter): ");
    scanf("%s", szoveg);        // space nem lehet benne
    viragnyelven_kiir(szoveg);
    return 0;
}
void viragnyelven_kiir(char sztring[]) {
    int i, j, maganh;
    // inicializálás karakterenként: a lezáró karaktert meg kell adni
    char mgh[6] = {'a', 'e', 'i', 'o', 'u', '\0'};
    // karaktertömb indexelése 0-tól kezdődik
    // kilépési feltétel a lezáró '\0' karakter elérése
```

```
    for (i=0; sztring[i]!='\0'; i++) {
        maganh = 0;
        for (j=0; mgh[j]!='\0'; j++)
       // mgh tömbelemek kisbetűsek -> a karaktereket kisbetűssé alakítjuk
            if ( tolower(sztring[i]) == mgh[j] ) maganh = 1;
        if (maganh) printf("%cv%c", sztring[i], tolower(sztring[i]));
        else printf("%c", sztring[i]);
    }
    return ;
}
```

10.2.2 Sztringkezelő és karakterkonverziós függvények

10.2.2.1 Feladat:

Olvassunk be a szabványos bemenetről Esc+Enter lenyomásáig karaktereket. Írjuk vissza megfordítva, nagybetűssé konvertálva és állapítsuk meg, hogy a megadott szöveg palindróma-e (megegyezik-e a fordítottjával).

```
#include <stdio.h>
#include <string.h>          // sztringkezelő függvényekhez
#include <ctype.h>           // karakterkonverziós fv-ekhez
#define  N  100
#define  ESC 27              // Esc billentyű ASCII kódja
char * szokoztorol(const char * sztring);
char * megfordit(const char * sztring);
char * nagybetusit(char * sztring);

int main() {
    char szoveg[N];
    int i=0;
    char ch;
    printf("Esc+Enter lenyomasaig olvassa a karaktereket: ");
    while ((ch=getchar()) != ESC) {     // lehet benne szókoz
        szoveg[i] = ch;
        i++;
    }
    szoveg[i] = '\0';            // a lezáró karaktert a végéhez kell fűzni
    char * nagybetus = nagybetusit(szoveg);
    printf("Visszafele nagybetusen: %s\n", megfordit(nagybetus));
    char * egybeirt = szokoztorol(nagybetus);  // szövegből sztring lesz
    // két sztring összehasonlítása (kis- és nagybetű érzékeny)
    if (strcmp(egybeirt, megfordit(egybeirt)) == 0)
        printf("A megadott szoveg palindroma.");
    else
        printf("A megadott szoveg nem palindroma.");
```

```
    return 0;
}

char * nagybetusit(char * sztring){
    char * seged = sztring;
    while(*sztring) {
        *sztring=toupper(*sztring);     // ctype.h standard függvénye
        sztring++;
    }
    return seged;
}

char * megfordit(const char * sztring) {
    char * forditott;
    char * seged = forditott;
    int i = strlen(sztring)-1;  // sztring hossza
    while (i>=0) {
        *forditott = sztring[i];
        forditott++;
        i--;
    }
    *forditott = '\0';          // a lezáró karaktert a végére kell fűzni
    return seged;               // a megfordított karaktertömb kezdőcíme
}

char * szokoztorol(const char * sztring) {
    char * seged;
    char * p = seged;
    int honnan, hova = 0;
    for (honnan=0; sztring[honnan] != '\0'; honnan++) {
        if (sztring[honnan] != ' ' ) {
            seged[hova]  = sztring[honnan];
            hova++;
        }
    }
    seged[hova] = '\0';     // lezáró karakter a végére
    return p;               // szóközök nélküli karaktertömb kezdőcíme
}
```

10.2.3 Sztringek tömbje

10.2.3.1 Feladat:

Kérjük be a felhasználó születési dátumát 'YYYY.MM.DD.' alakban karaktersorozatként és számítsuk ki az életkorát. A megoldáshoz a beolvasott sztringet szét kell bontani a megadott formátum szerint három egész számra. Ehhez a sztringből olvasó standard sscanf() függvény

használható. Az aktuális rendszerdátumot a __DATE__ makró tartalmazza 'MMM DD YYYY' alakban. Alakítsuk ezt át a megadott születési dátum formátumára, és így számoljunk.

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

char * aktualis_datum(void);
int hanyadikHonap(char * honap);

int main() {
    char szul_datum[12];
    char * mai_datum;
    int szul_ev, szul_honap, szul_nap, mai_ev, mai_honap, mai_nap;
    printf("Mikor szulettel (yyyy.mm.dd.) ? ");
    scanf("%s", szul_datum);
    // dátum szétbontása sztringből olvasással
    if(sscanf(szul_datum, "%d.%d.%d.",&szul_ev,&szul_honap,&szul_nap)!=3){
        printf("Hibas a datum!");
        exit(-1);
    }
    mai_datum=aktualis_datum();
    printf("A mai datum: %s\n", mai_datum);
    sscanf(mai_datum, "%d.%d.%d.", &mai_ev, &mai_honap, &mai_nap);
    // életkor számítása
    if (mai_honap >= szul_honap)  printf("%d eves %d honapos vagy.",
            mai_ev-szul_ev, mai_honap-szul_honap);
    else printf("%d eves %d honapos vagy.",
            mai_ev-szul_ev-1, 12-(szul_honap-mai_honap));
    return 0;
}

char * aktualis_datum(void) {
    int mai_ev, mai_nap;
    char mai_honap[4], mai_datum[12];
    // __DATE__ előredefiniált makró; aktuális dátum "MMM DD YYYY" alakban
    sscanf(__DATE__, "%s %d %d", mai_honap, &mai_nap, &mai_ev);
    int mai_honap_szammal = hanyadikHonap(mai_honap);
    sprintf(mai_datum, "%d.%d.%d.", mai_ev, mai_honap_szammal, mai_nap);
    return mai_datum;
}

int hanyadikHonap(char * honap) {
    char honapok[][4] = {"-", "Jan", "Feb", "Mar", "Apr", "May", "Jun",
                         "Jul", "Aug", "Sep", "Okt", "Nov", "Dec"};
    //vagy mutatótömbös alakban: char *honapok[] = { ... };
```

```
    int i;
    for(i=0; i<13; i++) if(strcmp(honap, honapok[i])==0)
            return i;         // ha a két sztring azonos
}
```

10.2.3.2 Feladat:

Készítsünk szókitaláló játékprogramot (akasztófa). Tömbben tároljunk szavakat, amelyek közül a számítógép véletlenszerűen választ egyet. A felhasználó egyesével adja meg a betűket és a számítógép megmutatja, hogy az(ok) hol található(k) a szóban.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
char * szo_valaszt(void);
char * szoveg_init(const char *forras, char *cel);

int main() {
    char betu;
    int talalat=0, i, kerdes=0;
    char * gondolt=szo_valaszt();          // a gondolt szó
    char kitalalt[strlen(gondolt)+1];      // kitalált karakterek tömbje
    char * seged = szoveg_init(gondolt, kitalalt);// _ jelekkel kitöltés
    printf("Felirtam egy szot. Talald ki! Kerdezz betuket!\n");
    printf("A gondolt szo: %s\n", seged);
    do {
        printf("Van benne? ");
        scanf(" %c", &betu);
        kerdes++;                    // számoljuk hány kérdésből találja ki
        for (i=0; i<strlen(gondolt); i++) {
            if (gondolt[i]==tolower(betu)) {
                kitalalt[i]=tolower(betu);
                talalat++;
            }
        }
        printf("%s\n", kitalalt);
    } while (strlen(gondolt)!=talalat);   // amíg van ki nem talált karakter
    printf("Gratulalok, %d kerdesbol kitalaltad!", kerdes);
    return 0;
}
char * szo_valaszt(void) {
    // mutatótömbös megvalósítás
  char * szavak[] = {"bagolyvar","balettruha","fuggonykarnis","hangyaboly",
      "kiskakas","krumpliorr","menyasszony","napszemuveg","szalmakalap",
      "tolltarto", NULL};//tömb bejárás kilépési feltételhez NULL az utolsó
    int db = 10;
```

```
    srand(time(0));
    return szavak[rand()%db];  //0 és tömbméret közötti index
}
char * szoveg_init(const char *forras, char *cel) { //_ jelekkel feltöltés
    char *seged = cel;
    while (*forras) { *cel = '_'; cel++; forras++; }
    *cel = '\0';
    return seged;
}
```

10.2.4 Integerként kezelt sztringkonstansok (enum adattípus)

10.2.4.1 Feladat:

Írjuk meg a kő-papír-olló játék kódját. Számoljuk, hogy ki hányszor nyert. A választást (kő, papír, olló) és a nyertest (senki, felhasználó, gép) enum típusként definiáljuk. A felsorolt adattípus sztringkonstansokat tartalmaz, amelyeknek a fordító által feldolgozott értéke egy egész szám: alapértelmezés szerint a megadott sorrendben 0-tól kezdve egyesével növekvő értékek. A sztringkonstansok szerepe tehát formális: a kód olvashatóságát javítják.

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

int ki_nyert (int Fvalasz, int Gvalasz);
int beolvas (int * Fvalasz);

typedef enum {KO, PAPIR, OLLO} valasz_tipus;
typedef enum {SENKI, FELH, GEP} jatekos_tipus;

int main() {
  int jo, fpont = 0, gpont = 0, jatek = 0;
  char ujra;
  srand(time(0));
  valasz_tipus Fvalasztas, Gvalasztas;
  jatekos_tipus nyertes;

  printf("Jatsszunk ko-papir-ollo-t!\n");
  do {
    do { jo=beolvas(&Fvalasztas); } while (!jo); // ellenőrzött beolvasás
    Gvalasztas = rand()%2;           // 0, 1, vagy 2
    printf("Gep: %d\n", Gvalasztas);
    jatek++;
    nyertes = ki_nyert(Fvalasztas, Gvalasztas);
    switch (nyertes) {
      case FELH:
        printf("Te gyoztel!\n"); fpont++;
```

```
        break;
      case GEP:
        printf("En gyoztem!\n"); gpont++;
        break;
      default:
        printf("Dontetlen!\n");
    }
    printf("Akarsz meg jatszani (i/n)? ");
    scanf(" %c", &ujra);              // %c előtt space van!!!
  } while (ujra=='i' || ujra=='I');
  printf("Osszesites: %d-bol %d-szor nyertel.", jatek, fpont);
  return 0;
}

int beolvas (int * Fvalasz) {
  printf("Mit valasztasz? 0-ko, 1-papir, 2-ollo: ");
  if (scanf("%d", Fvalasz)!=1) {
    printf("Rossz valasz!\n");
    exit(-1);
  }
  else if ( *Fvalasz != 0 && *Fvalasz != 1 && *Fvalasz != 2 ) {
    printf("Rossz valasz!\n");
    return 0;
  }
  return 1;
}

int ki_nyert (int Fvalasz, int Gvalasz) {
  if (Fvalasz == Gvalasz) return SENKI;
  switch (Fvalasz) {
    case KO :
      if (Gvalasz==PAPIR) return GEP;
      else return FELH;
      break;
    case PAPIR :
      if (Gvalasz==OLLO) return GEP;
      else return FELH;
      break;
    case OLLO :
      if (Gvalasz==KO) return GEP;
      else return FELH;
      break;
  }
}
```

11 Kétdimenziós tömbök, dinamikus helyfoglalás

11.1 Elméleti alapok

A tömb dimenzióinak számát és azok méretét deklaráláskor adjuk meg: a tömb neve után kitett szögletes zárójelpárok száma jelzi a dimenziószámot és a zárójelekben szereplő értékek az egyes dimenziók méretei. C-ben a dimenziók számára vonatkozóan nincs elvi korlátozás. Ebben a fejezetben a kétdimenziós tömbök használatát gyakoroljuk. A kétdimenziós tömb az elfoglalt memóriaterület kezdőcímét kijelölő mutatóként adódik át függvénynek argumentumként (ugyanúgy, mint az egydimenziós tömb). A hívott függvényben a kétdimenziós tömb (mátrix) elemeire hivatkozás formája: mx[i*oszlop+j].

Dinamikus helyfoglaláskor a program futása során allokálunk memóriaterületet. Erre akkor van szükség például, ha fordítási időben még nem ismert egy tömb mérete; vagy ha függvényen belül lokális változóként deklarálunk tömböt, amit a függvényből való visszatérés után is szeretnénk elérni. Ugyanis a dinamikusan foglalt terület nem a verem, hanem a kupac memóriaszegmensben található. A dinamikusan foglalt memóriaterület felszabadítása a programozó feladata.

11.2 Gyakorló feladatok

11.2.1 Kétdimenziós numerikus tömbök

11.2.1.1 Feladat:
Írjunk C programot, amely külön függvényekben valósítja meg két mátrix összeadását, illetve szorzását. Az egyik mátrixot inicializáljuk, a másik elemeit kérjük be a felhasználótól. Ellenőrizzük a műveletek elvégezhetőségét! Két mátrix akkor összegezhető, ha a sorok és az oszlopok száma rendre azonos. Összeszorzásuk akkor végezhető el, ha az első szorzótényező mátrix oszlopainak száma megegyezik a második sorainak számával.

```
#include <stdio.h>
#include <stdlib.h>     //dinamikus memóriakezelő fv-ekhez

void matrix_beolvas(int *mx, int sor, int oszlop) {
    int i, j;
    for(i=0; i<sor; i++) {
        for(j=0; j<oszlop; j++) {
                printf("%d. sor %d. oszlop: ", i+1, j+1);
                scanf("%d", &mx[i*oszlop+j]);
```

```
        }
    }
    return ;
}

void matrix_kiir(int *mx, int sor, int oszlop) {
      int i, j;
      for(i=0; i<sor; i++) {
            for(j=0; j<oszlop; j++)
                  printf("%3d ", mx[i*oszlop+j]);
            printf("\n");
      } printf("\n");
      return ;
}

int * matrix_osszeadas(int *mx1, int *mx2, int sor, int oszlop) {
      int i, j;
      int * e;
      /*lefoglaljuk az eredménymátrixnak megfelelő méretű területet*/
      /*malloc: a lefoglalt memóriaterület inicializálatlan*/
      e=(int *)malloc(sor*oszlop*sizeof(int));
      if(e==NULL){
            printf("Nincs elég memória!");
            exit(-1);
      }
      for(i=0; i<sor; i++)
            for(j=0; j<oszlop; j++)
                  e[i*oszlop+j]= mx1[i*oszlop+j] + mx2[i*oszlop+j];
      return e;
}

int * matrix_szorzas(int * a, int sor1, int oszlop1, int * b, int sor2, int
oszlop2) {
    int * e;
    int i,j,k,sum;
    //lefoglaljuk az eredménymátrixnak megfelelő méretű területet
    //calloc:int esetén használható, a lefoglalt terület 0-val inicializált
    e=(int *)calloc(sor1*oszlop2, sizeof(int));
    if(e==NULL){
        printf("Nincs elég memória!");
        exit(-1);
    }
    for(i=0;i<sor1;i++)
        for(j=0;j<oszlop2;j++){
            sum=0;
            //vagy for(k=0; k<sor2; k++)
```

```
            for(k=0;k<oszlop1;k++){
                sum+=a[i*oszlop1+k]*b[k*oszlop2+j];
            }
            e[i*oszlop2+j]=sum;
        }
    return e;
}

int main(){
    int matrix1[3][2] = { {1, 2},
                          {3, 4},
                          {5, 6} };
    int matrix2[3][2] = { {1, 1},
                          {1, 1},
                          {1, 1} };
    int matrix3[2][4];
    int s1=3, o1=2, s2=3, o2=2, s3=2, o3=4; //a mátrixok mérete
    int *eredmeny;
    if(s1!=s2 || o1!=o2)        //műv. elvégezhetőségének ellenőrzése
        printf("Nem osszeadhato matrixok!");
    else {
        eredmeny=matrix_osszeadas((int*)matrix1,(int*)matrix2,s1,o1);
        printf("A ket inicializalt matrix osszege:\n");
        matrix_kiir((int*)eredmeny,s1,o1);
    }
    printf("%dx%d matrix elemeinek beolvasasa:\n", s3, o3);
    matrix_beolvas((int*)matrix3,s3,o3);
    if(o1!=s3)                  //műv. elvégezhetőségének ellenőrzése
        printf("Nem osszeszorozható matrixok!");
    else {
        eredmeny=matrix_szorzas((int*)matrix1,s1,o1, (int*)matrix3,s3,o3);
        printf("Az elso es a harmadik matrix szorzata:\n");
        matrix_kiir((int*)eredmeny,s1,o3);
    }
    free((int*)eredmeny);//dinamikusan lefoglalt mem.terület felszabadítása
    return 0;
}
```

11.2.1.2 Feladat:

Hozzunk létre dinamikusan egy 10x10-es mátrixot és töltsük fel a főátló elemeit 1 és 10 közé eső véletlenszámokkal, a többit pedig 0-val (diagonálmátrix). Számítsuk ki külön függvénnyel a mátrix determinánsát (a főátló elemeinek szorzata).

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

```c
#include <time.h>

void matrix_kiir(int *mx, int sor, int oszlop) {
      int i, j;
      for(i=0; i<sor; i++) {
            for(j=0; j<oszlop; j++)
                  printf("%3d ", mx[i*oszlop+j]);
            printf("\n");
      }
      printf("\n");
      return ;
}


void diagonalmx_feltolt(int *mx, int sor, int oszlop) {
    int i, j;
    for(i=0; i<sor; i++) {
        for(j=0; j<oszlop; j++) {
            if (i==j)                              /*főátló elemei*/
                mx[i*oszlop+j] = rand()10+1;    /*1...10*/
            else
                mx[i*oszlop+j] = 0;
        }
    } return ;
}

long diagonalmx_det(int *mx, int sor, int oszlop) {
    int i, j;
    long det=1;
    for(i=0; i<sor; i++)
        for(j=0; j<oszlop; j++) {
            if(i==j)
                det *= mx[i*oszlop+j];
            else
                continue;
        }
    return det;
}

int main(){
    int *diagonalmx;
    long determinans;
    /*calloc: 0-val inicializálja a lefoglalt memóriaterületet */
    diagonalmx = (int*)calloc(10*10, sizeof(int));
    if(diagonalmx==NULL) {
        printf("Nincs eleg memoria");
```

```
        exit(-1);
    }
    srand(time(0));
    diagonalmx_feltolt((int*)diagonalmx,10,10);
    matrix_kiir((int*)diagonalmx,10,10);
    determinans = diagonalmx_det((int*)diagonalmx,10,10);
    printf("A matrix determinansa: %ld\n",determinans);
    free((int*)diagonalmx);   //dinamikusan foglalt memória felszabadítása
    return 0;
}
```

11.2.2 Kétdimenziós karaktertömbök

11.2.2.1 Feladat:

Készítsünk C programot, amely kiírja az aktuális dátumot és naptári napot. Ehhez szükségünk van egy kiinduló pontra. Legyen most 1900. január 1. hétfő. Meg kell határozni az azóta eltelt napok számát. A naptári napokat sztringtömbben tároljuk, de csak rövidítve írjuk ki (az első három karaktert). Sztringtömb első indexe egy sztringet, a második index a sztring egy karakterét jelöli ki.

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

int szokoev_e(int ev);
char * aktualis_datum(void) {...}          // Lásd 10.2.3.1 feladat
int hanyadikHonap(char * h) {...}          // Lásd 10.2.3.1 feladat

int main( ) {
    int honap[] = {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
  /*mutatótömb: char* napok[]={"Sunday", "Monday", "Tuesday", "Wednesday",
    "Thursday", "Friday", "Saturday"};*/
    char napok[][10] = {"Sunday", "Monday", "Tuesday", "Wednesday",
       "Thursday", "Friday", "Saturday"};
    int n, nap, ho, ev, i;

    char * mai_datum=aktualis_datum();
    sscanf(mai_datum, "%d.%d.%d.\n", &ev, &ho, &nap);
    n=nap;
    // szökőévben február 29 napos
    if (szokoev_e(ev)) honap[2] = 29;
    // hanyadik napja az évnek
    for (i = 1; i < ho; i++)
        n += honap[i];
    // 1900.jan.1.(hétfő) óta eltelt napok száma
    for (i = 1900; i < ev; i++)
```

```
        if (szokoev_e(i)) n += 366;
        else n += 365;
    // rövidítve írjuk ki a napot
    /* sztringtömb első indexe egy sztringet, második indexe a sztring egy
      karakterét jelöli ki */
    printf("%d-%d-%d: %c%c%c \n",
            ev, ho, nap, napok[n%7][0], napok[n%7][1], napok[n%7][2]);
    return 0;
}


int szokoev_e(int ev) {
  if (ev%400 == 0) return 1;
  else if (ev%100 == 0) return 0;
  else if (ev%4 == 0) return 1;
  else return 0;
}
```

11.2.2.2 Feladat:

Írjuk meg a 'gondoltam egy számot' játék kódját. A számítógép véletlenszerűen előállít egy 1 és 10 közötti számot és menüből választható, előre rögzített kérdések megválaszolásával segít kitalálni a számot. A felhasználónak legyen lehetősége többször egymás után játszani. A menüt valósítsuk meg kétdimenziós karaktertömb (sztringtömb) és egydimenziós mutatótömb formájában is.

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

int menukiir(void);
int prim_e(int szam);

int main()
{
    char ujra;
    int gondolt, tipp, valasztas, vege;
    srand(time(0));
    do {
        gondolt=rand()%10+1;
        vege=0;
        printf("Gondoltam egy szamot 1 es 10 kozott. Kerdezz!");
        do {
            valasztas = menukiir();
            switch (valasztas) {
                case 1 :
```

```
                    if (gondolt%2==0) printf("Igen.");
                    else printf("Nem.");
                    break;
                case 2:
                    if (prim_e(gondolt)) printf("Igen.");
                    else printf("Nem.");
                    break;
                case 3:
                    if (gondolt>5) printf("Igen.");
                    else printf("Nem.");
                    break;
                case 4:
                    printf("Tipp: "); scanf("%d", &tipp);
                    if (tipp==gondolt) {
                        printf("Gratulalok, kitalaltad!");
                        vege = 1;
                    }
                    else printf("Sajnos nem talalt.");
                    break;
            }
        } while (!vege);
        printf("\nAkarsz meg jatszani (i/n)? ");
        scanf(" %c", &ujra);            // %c előtt space van!!!
    } while (ujra=='i' || ujra=='I');
    return 0;
}

int menukiir(void) {
    int i, valasz;
    /* // Sztringtömbös megvalósítás
    char menu[][20]= { "\n", "1. Paros szam?",
                       "2. Prim szam?",
                       "3. 5-nel nagyobb?",
                       "--------------",
                       "4. Tippelek", "\n" };

    for(i=0;i<7;i++) printf("%s\n", menu[i]);
    */

    // Mutatótömbös megvalósítás
    char *menu[]= { "\n", "1. Paros szam?",
                       "2. Prim szam?",
                       "3. 5-nel nagyobb?",
                       "--------------",
                       "4. Tippelek", "\n", NULL };
```

```
    char **p;  p=menu;
    while(*p) { printf("%s\n", *p); *p++; }
    //---------------------------

    if (scanf("%d", &valasz) != 1) {
        printf("Rossz valasz!");
        exit(-1);
    }
    switch(valasz) {
        case 1:
            return 1; break;
        case 2:
            return 2; break;
        case 3:
            return 3; break;
        case 4:
            return 4; break;
        default:
            printf("Nincs ilyen menupont!"); return 0;
    }
}

int prim_e (int szam) {
    int oszto;
    for (oszto=2; oszto<=szam/2; oszto++) {
        if (szam%oszto==0) return 0;         //nem prím
    }
    return 1;                                //prím
}
```

12 Struktúrák

12.1 Elméleti alapok

A struktúra a C nyelvben összetett adattípus. Függvény és void típus kivételével tetszőleges típusú adattagjai a memóriában szomszédosan helyezkednek el. Deklarációja:

```
struct tipusnev                          //a struktúra típus neve
{
        tipus adattag1;
        tipus adattag2;
        ...
} strukturavaltozo, *strukturamutato;
```

Struktúraváltozónak és -mutatónak értéket adni vagy adattagonként lehet, vagy az = operátorral. Adattagonkénti értékadásnál az adattagok kijelölésére struktúraváltozó esetén a . operátor, struktúra mutató esetén a → operátor szolgál. Az = operátorral azonos szerkezetű struktúra adattagjai másolhatók át egy utasításban. A következő szakaszok a struktúrák használatának tipikus eseteire mutatnak példákat.

12.2 Gyakorló feladatok

12.2.1 Új típus létrehozása

12.2.1.1 Feladat:

Kernighan&Ritchie: A C programozási nyelv (ANSI C) című könyvben az 5.7 Többdimenziós tömbök fejezetben található dátumkonverziós példa alapján.

1. Készítsen dátum típust definiáló struktúrát: struct date {int year; int month; int day;}; .
2. Írjon függvényt, amely egy 'yyyy.mm.dd.' formátumban megadott dátum sztringet átalakít úgy, hogy a hónap betűvel legyen kiírva.
3. Írjon függvényt, amely egy megadott dátumról eldönti, hogy hányadik napja az évnek.

```
#include <stdio.h>

typedef struct date {                   //struktúra típus deklarálása
    int year, month, day;
} Date;                                 //a típus rövid neve

void datumBeolvas(Date * d);
char * honapNeve(int n);
int hanyadikNap(Date d);
int szokoev_e(int ev) {...}             //Lásd 11.2.2.1 feladat

/* napok száma hónaponként; a tömb 0. indexű elemét nem használjuk*/
char napok_szama[2][13] = {
     {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
     {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31} //szökőévben
};

int main() {
    int evnapja;
    char * honap;
    printf("Adjon meg egy datumot YYYY.MM.DD. formaban: ");
    Date d;
    datumBeolvas(&d);
    honap = honapNeve(d.month);           //hónap átalakítása
```

```
    printf("%d. %s %d.\n", d.year, honap, d.day);
    evnapja=hanyadikNap(d);              //nap kiszámítása
    printf("Az ev %d. napja.\n", evnapja);
    return 0;
}

void datumBeolvas(Date * d){
    char str[12];
    scanf("%s", str);
    if (sscanf(str, "%d.%d.%d.", &d->year, &d->month, &d->day) != 3){
        printf("Hibas a datum!");
        exit(-1);
    }
    return ;
}

char * honapNeve(int n) {
    char* honapok[ ]={"?", "Januar", "Februar", "Marcius", "Aprilis",
      "Majus", "Junius", "Julius", "Augusztus", "Szeptember", "Oktober",
      "November", "December"};
    return (n<1 || n>12) ? honapok[0] : honapok[n];
}

int hanyadikNap(Date d) {
    int i, hanyadik=d.day, szokoev=0;
    if (szokoev_e(d.year)) szokoev=1;     //szökőév meghatározása
    for (i=1; i<d.month; i++)
      hanyadik += napok_szama[szokoev][i];
    return hanyadik;
}
```

12.2.1.2 Feladat:

Hozzon létre struktúrát egy kör adatainak tárolásához: x és y koordinátákkal adott középpont és sugár; ahol a középpont külön struktúra. Írjon függvényeket:

1. A kör adatainak beolvasására.
2. Egy beolvasott tetszőleges pontról döntse el, hogy rajta van-e a körvonalon (a kör középpontjától mért távolsága egyenlő-e a kör sugarával). Két pont távolsága a Pitagorasz-tétel alapján: sqrt($(kx-px)^2 + (ky-py)^2$).

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

typedef struct pont {
    double x, y;
} Pont;
```

```
typedef struct kor {
    Pont kp;
    double r;
} Kor;

/* névtelen struktúra beágyazással */
//struct circle {int r; struct{int x; int y;} center;};

void korBeolvas(Kor * k);
void pontBeolvas(Pont * p);
int korPontja_e(Kor k, Pont p);

int main(void) {
    Kor k;
    Pont p;
    korBeolvas(&k);
    printf("Kerem a pont koordinatait: ");
    pontBeolvas(&p);
    if (korPontja_e(k,p)) printf("Rajta van a korvonalon.");
    else printf("Nincs a korvonalon.");
    return 0;
}

void korBeolvas(Kor * k) {
    printf("Adja meg a kor kozeppontjat <x,y>: ");
//k: struktúrára mutató pointer, adattagjának kijelölése -> operátorral
    scanf("%lf, %lf", &k->kp.x, &k->kp.y);
    printf("Adja meg a kor sugarat: ");
    scanf("%lf", &k->r);
    return ;
}
```

12.2.2 Struktúrát visszaadó függvények

12.2.2.1 Feladat:

Készítsen C programot a másodfokú egyismeretlenes egyenletek megoldásához struktúrák használatával. Az ilyen egyenletek általános alakja: $\mathbf{ax^2 + bx + c = 0}$, ahol a, b és c konstansok.
Ha a=0 de b és c nem 0, az egyenlet elsőfokú, megoldása: x = -c/b.
Ha c=0 de a és b nem 0, az egyenlet hiányos, megoldásai: x1 = 0, x2 = -b/a.
Ha b=0 de a és c nem 0, az egyenlet tiszta másodfokú, megoldásai: x1,x2 = +- sqrt(-c/a)
Ha mindhárom konstans 0, akkor bármely szám megoldás.
Ha a és b 0, viszont c nem, akkor nincs megoldás.
A megoldóképletben a gyök alatti kifejezés (b^2-4ac) a másodfokú egyenlet diszkriminánsa. Ha

d>0, az egyenletnek két valós gyöke van. Ha d=0, az egyenletnek egy valós gyöke van (x1=x2). Ha d<0 az egyenletnek két komplex gyöke van.

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

typedef struct valosgyok {     //valosgyok típus deklarálása
    double x1;
    double x2;
} Valosgyok;

typedef struct komplex {       //komplex típus deklarálása
    double valos;
    double kepzetes;
} Komplexgyok;

Valosgyok valosgyok_szamitas(int, int, int, double);
Komplexgyok komplexgyok_szamitas(int, int, int, double);

int main() {
    int a, b, c;
    double d;
    printf("\nKerem a masodfoku egyenlet konstansait: ");
    scanf("%d, %d, %d", &a, &b, &c);
    if (a == 0) {
        if (b == 0) {
            if (c == 0)
                printf("\nBarmely szam megoldas.\n");
            else
                printf("\nNincs megoldas.\n");
        }
        else
        printf("\nAz egyenlet elsofoku, megoldasa: %.2lf\n", -c/(double)b);
    }
    else {
        d = b*b-4*a*c;                  //diszkrimináns számítása
        if (d>=0) {
        Valosgyok valosmegoldas = valosgyok_szamitas(a,b,c,d);
        printf("\nAz egyenlet valos gyokei: %.2lf, %.2lf\n",
             valosmegoldas.x1, valosmegoldas.x2);
        }
        else {
        Komplexgyok komplexmegoldas = komplexgyok_szamitas(a,b,c,d);
       printf("\nAz egyenlet komplex gyokei: %.2lf+%.2lfi,%.2lf-%.2lfi\n",
            komplexmegoldas.valos, komplexmegoldas.kepzetes,
            komplexmegoldas.valos, komplexmegoldas.kepzetes);
```

```
        }
    }
    return 0;
}


/* Másodfokú egyenlet valós gyökeinek meghatározása */
Valosgyok valosgyok_szamitas(int a, int b, int c, double d) {
    Valosgyok eredmeny;
    if (d == 0)
       eredmeny.x1 = eredmeny.x2 = -b/(double)(2*a);  //nem egészosztás!
    else if (d > 0) {
       eredmeny.x1 = ((-b)+sqrt(d))/(2*a);      //műv. sorrend kijelölése
       eredmeny.x2 = ((-b)-sqrt(d))/(2*a);
    }
    return eredmeny;
}

/* Másodfokú egyenlet komplex gyökeinek meghatározása */
Komplexgyok komplexgyok_szamitas(int a, int b, int c, double d) {
    Komplexgyok eredmeny;
    d = -d;
    eredmeny.valos = (-b)/(double)(2*a);
    eredmeny.kepzetes = sqrt(d)/(2*a);
    return eredmeny;
}
```

12.2.3 Struktúra (különböző típusú adatok halmaza) és struktúratömb.

12.2.3.1 Feladat:

CD-kről a következő adatokat szeretnénk nyilvántartani: cím, előadó, ár, megjelenés éve. Deklaráljon struktúrát ezen adatok tárolására. Írjon függvényt egy CD adatainak beolvasására és kiírására, valamint árának módosítására.

```
#include <stdio.h>

typedef struct CD {
      char cim[40];
      char eloado[40];
      double ar;
      int ev;
} CD;

void beolvas(CD * lemez);
```

```
void kiir(CD lemez);
int modosit(CD * lemez);

int main()  {
    CD lemez; beolvas(&lemez);
    int siker=modosit(&lemez);    //módosításhoz címszerint kell átadni
    if (siker) printf("\nA CD árát 10%%-al növeltem.\n");
    return 0;
}

void beolvas(CD * lemez) {
    printf("CD cime: ");
    scanf("%s", lemez->cim);          /*space nem lehet a sztringben*/
    printf("CD eloadoja: ");
    scanf("%s", lemez->eloado);       /*tömb adattag esetén nincs &*/
    printf("CD ara: ");
    scanf("%lf", &lemez->ar);         /*az ár adattag int, ezért kell &*/
    printf("CD megjelenesi eve: ");
    scanf("%d", &lemez->ev);
    return ;
}

void kiir(CD lemez) {
    printf("\nCD cime: %s", lemez.cim );
    printf("\nCD eloadoja: %s", lemez.eloado );
    printf("\nCD ara: %.2f", lemez.ar );
    printf("\nCD megjelenesi eve: %d\n", lemez.ev );
    return ;
}

/* CD árának 10%-os növelése */
int modosit(CD * lemez) {
    lemez->ar = lemez->ar * 1.1;
    return 1;
}
```

12.2.3.2 Feladat:

Az előző feladat alapján hozzon létre egy 10 elemű CD tárat, töltse fel adatokkal. Egy struktúratömbbe gyűjtse ki a 2010 után megjelent CD-ket. Mindkét tömbnek dinamikusan foglaljon helyet.

```
#include <stdio.h>
typedef struct CD {          /* CD típus deklarálása */
      char cim[40];
      char eloado[40];
```

```
      double ar;
      int ev;
} CD;

void beolvas(CD * lemeztar, int meret);
void kiir(CD talalat[], int meret);
int kivalogat(CD lemeztar[], int meret, int mikortol, CD * talalat);

int main()  {
    int i, db;
    printf("Hány elemű legyen a CD tár? Válasz: "); scanf("%d", &db);
    CD * cdtar = (CD *)malloc(sizeof(CD)*db);  //dinamikus helyfoglalás
    if (cdtar == NULL) {
            printf("\nNincs eleg memoria!\n");
            exit(-1);
    }
    beolvas(cdtar, db);

    /*Adott év után megjelent CD-k kiválogatása*/
    CD * valogatas = (CD *)malloc(sizeof(CD)*db);  //dinamikus helyfoglalás
    if (valogatas == NULL) {
            printf("\nNincs eleg memoria!\n");
            exit(-1);
    }
    int mikortol = 2010;
    int t_db = kivalogat(cdtar, db, mikortol, valogatas);
    if (t_db) {                                    /* ha nem üres a lista */
        printf("\n\nA %d után megjelent CD-k listája: ", mikortol);
        kiir(valogatas, t_db);
    }
    else printf("\n\nNincs %d után megjelent CD.\n", mikortol);
    free(cdtar);            // dinamikusan foglalt memória felszabadítása
    free(valogatas);
    return 0;
}

void beolvas(CD * lemeztar, int meret) {
      int i;
      for (i=0; i<meret; i++)  {
        printf("\nKerem a(z) %d. CD cimet: ", i+1);
        scanf("%s", lemeztar[i].cim);      /*space nem lehet a sztringben*/
        printf("Kerem a(z) %d. CD előadóját: ", i+1);
        scanf("%s", lemeztar[i].eloado);  /*tömb adattag esetén nincs &*/
        printf("Kerem a(z) %d. CD arat: ", i+1);
        scanf("%lf", &lemeztar[i].ar);
        printf("Kerem a(z) %d. CD megjelenesi evet: ", i+1);
```

```
        scanf("%d", &lemeztar[i].ev);
    }
    return ;
}
void kiir(CD talalat[], int meret) {
    int i;
    for (i=0; i<meret; i++)  {
        printf("\n%d. CD: ", i+1);
        printf("\nCD cime: %s", talalat[i].cim );
        printf("\nCD eloadoja: %s", talalat[i].eloado );
        printf("\nCD ara: %.2f", talalat[i].ar );
        printf("\nCD megjelenesi eve: %d\n", talalat[i].ev );
    }
    return ;
}

int kivalogat(CD lemeztar[], int meret, int mikortol, CD * talalat) {
    int i, db=0;
    for (i=0; i<meret; i++) {
        if (lemeztar[i].ev >= mikortol) {
            //értékadás struktúratömb elemének az adattagok átmásolásával
            talalat[db] = lemeztar[i];
            db++;
        }
    }
    return db;
}
```

Struktúra adattagja lehet saját típusával megegyező típusú objektumra mutató pointer. Az ilyen struktúrákat önhivatkozónak nevezzük és láncolt listák építésére használjuk.

13 Fájlkezelés

Ebben a fejezetben azt mutatjuk be, hogyan kezelünk fájlokat C nyelven. Ha szeretnéd elmenteni, betölteni az adataidat a programodban, akkor ezeket kell megismerned.
Mi is a fájl? A fájl bájtok egy olyan csoportja, amit jellemzően valamilyen háttértárolón tárolunk. Hogy hogyan értelmezzük ezt a bájtgyűjteményt az alapvetően csak rajtunk múlik. Lehetnek ezek sima karakterek, szavak, bekezdések, szöveges oldalak. De lehetnek egy adatbázis mezői, táblái. Vagy egy grafikus kép pixelei. Csak rajtad, és a programodon múlik, hogyan értelmezed.

13.1 ASCII szövegfájlok

A karakterek kódolásának egy bevett módja az ASCII kódtábla. (Ezt most nem magyarázzuk el, nézz utána!) Fogjuk fel az ASCII fájlokat úgy, mintha ASCII kódolt karakterfolyamot tartalmaznának. Az ilyen fájlokat csakis elejétől a végéig, a karaktereket egymás után haladva dolgozhatjuk fel. Nyugi, a C programnyelv azért nem lesz ennyire korlátos, vannak olyan könyvtári függvények, amelyekkel egész sorokat is feldolgozhatunk.
Ha megnyitjuk ezt a fájlt, akkor el kell döntenünk, milyen műveletet akarunk végrehajtani: írást, olvasást vagy hozzáfűzést. És ezekből egyszerre csak egyet!

13.2 Bináris fájlok

A bináris fájlok nem sokban különböznek a szövegfájloktól. Ugyanúgy bájthalmazok. És ugye tudod, hogy C nyelven a bájt és a karakter ugyanazt a 8 bites adatstruktúrát jelenti. Akkor mégis mi a különbség? Bináris fájlokat teljesen tetszőleges módon hozhatunk létre, olvashatunk, írhatunk. Csak a programozón múlik.
Bináris fájlokban az adatot szekvenciálisan vagy össze-vissza ugrálva is elérhetjük. Ez C nyelvű programmal azt jelenti, hogy az aktuális fájl pozíciót a megfelelő helyre mozgatjuk, mielőtt írjuk vagy olvassuk az adatokat. Ez a bináris fájlkezelés jellemzője.
Bináris fájlokon egyszerre végezhetünk írási és olvasási műveleteket.

13.3 A fájlok megnyitásának módjai

Épp itt az ideje, hogy lekódoljuk az első fájlba író kódunkat

```
FILE *fp = fopen("file.txt", "w");
if (fp == NULL)
{
    printf("Hiba a fájl megnyitásakor!\n");
    exit(-1);
}
const char *text = "Hello World!";
fprintf(fp, "%s\n", text);
fclose(fp);
```

Az fopen függvény megnyitja a fájlt, az fclose pedig bezárja. Könnyű lesz megjegyezni, a kezdőbetű f, mint fájl. Milyen paramétereket várnak a függvények?

```
FILE *fopen(const char *fajl_nev, const char *megnyitas_mod);
```

A fájlnév az első paraméter, amikor megadod vigyázz arra, hogy minden speciális karaktert, amelyik szerepel a fájlnévben azt megfelelően kódolni kell. például a backslash ’\’ karaktert.
A megnyitás módja a következő lehet:

r (reading) olvasás
w writing) írás
a (append) hozzáfűzés
r+ írás + olvasás, az elején kezdünk
w+ írás+olvasás, felülírjuk a fájlt
a+ írás + olvasás, hozzáfűzünk

Ezek szöveges fájlokra vonatkoznak. Bináris fájloknál a betűk után még egy b betűt is oda kell írni, pl.: ”rb” vagy ”wb+”.
Ha a fájl megnyitása sikerült egy FILE struktúrára mutató pointert kapunk vissza, ha nem sikerült, akkor NULL pointert. Az alábbi példáimban fp lesz a FILE struktúrára mutató pointer neve.
Az fclose függvény bezárja a fájlt, amelyet paraméterként megkap:

```
int fclose(FILE *fp);
```

Ha sikerült, akkor a visszatérési érték nulla.

13.4 Írás és olvasás fájlokkal

Az fprintf, fscanf függvények írják és olvassák a fájlt. Használatuk megegyezi a képernyőre író és arrol olvasó printf, scanf függvényekkel, de az első paraméterük az fopen függvénnyel megnyitott FILE* pointer
Ha egyszerre csak egy karaktert szeretnénk írni akkor az fputs függyvényt kell használnunk:
int fputc(int c, FILE *fp);
Itt vigyáznunk kell, hogy c az unsigned char tartományba essen, azaz 0-255 közé. Ha sikerült az írás, a függvény visszatérési értéke c. Ha nem, akkor a visszatérési érték EOF (end of file, fájlvége).
Az én fejlesztőkörnyezetem ezt így definiálta:

```
#define EOF     (-1)
```

A függvény párja az

```
int fgetc (FILE *fp);
```

Ez egy karaktert (tudod: 0-255 közötti számot) olvas. Ha már nincs mit olvasnia, mert például elértük a fájl végét, akkor a visszatérési érték EOF.

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

```c
int main()
{
  float f1, f2;
  int i1, i2;
  FILE *fp;
  char my_filename[] = "szamok.txt";

  fp = fopen (my_filename, "w");
  fprintf (fp, "%f %f %d %d", 23.5, -12e6, 100, 5);

  /* Lezarjuk a fajlt, most nincds hibakezeles */
  fclose (fp);

  fp = fopen (my_filename, "r");
  fscanf (fp, "%f %f %i %i", &f1, &f2, &i1, &i2);

  /* Megint lezarjuk a fajlt */
  fclose (fp);

  printf ("Float 1 = %f\n", f1);
  printf ("Float 2 = %f\n", f2);
  printf ("Integer 1 = %d\n", i1);
  printf ("Integer 2 = %d\n", i2);

  return 0;
}
```

Ez valami ilyesmit ír ki a kimenetre:

```
Float 1 = 23.500000
Float 2 = -12000000.000000
Integer 1 = 100
Integer 2 = 5
```

25. ábra Kimenet

Azt írtam, hogy vannak olyan C nyelvi függvények, amelyek nem csak egy karaktert olvasnak be a szövegfájlból. Ime:

```c
char *fgets(char *str, int n, FILE *fp)
```

Ez a függvény az str nevű karaktertömbbe olvas be karaktereket. Ha tud, akkor n karaktert olvas be, de ha újsor karaktert vagy a fájl végét hamarabb eléri, akkor ott megáll. Ha sikeres az olvasás, a visszatérési érték is ugyanaz, mint az str. Ha nem sikerült, akkor az str változatlan marad és NULL pointert kapunk vissza.

Példa

```
#include <stdio.h>

int main()
	{
	FILE *fp;
	char str[255];

	/* fajl megnyitas olvasasra */
	fp = fopen("file.txt" , "r");
	if(fp == NULL) {
		return(-1);
		}
	if( fgets (str, 255, fp)!=NULL ) {
		/* amit beolvastam, kiirom a kepernyore */
		puts(str);
		}
	fclose(fp);

	return(0);
	}
```

Miben különbözik a következő kód:

```
#include <stdio.h>
int main( )
	{
	FILE *fp;
	char c;
	fp = fopen("file.txt", "r");
	if (fp == NULL)
		{
		return (-1);
		}
	do
		{
		c = getc(fp); /* egy karaktert beolvasunk  a fajlbol */
		putchar(c); /* kiirjuk a kepernyore */
		} while (c != EOF); /* amig a fajl vegre nem erunk */
	fclose(fp);
	return 0;
	}
```

13.5 Bináris fájlok írása és olvasása

Bináris fájlok írása és olvasása a

```
size_t fwrite(const void *ptr, size_t meret, size_t darab, FILE *fp);
size_t fread(void *ptr, size_t meret, size_t darab, FILE *fp);
```

függvényekkel történik. Az első paraméter a ptr a memória tetszőleges részére mutathat. Mivel void* a típusa ezért bármilyen adattípust elfogad, legyen az int *, float * vagy akár valamilyen struct (struktúra) pointere. A második paraméter az a bájtokban mért méret. Ha az első paramétert egy vektornak tekintjük, akkor minden vektorelem ilyen méretű. Fogadjunk, hogy nem tudod, hány bájtos egy int vagy egy float. Sebaj. a sizeof (*type-name*) függvény pont erre jó, majd az kiszámolja bármilyen adattípus, struktúra méretét. A harmadik paraméter az írni/olvasni kívánt vektorelemek száma. A negyedik pedig a fájl.

```
#include<stdio.h>

/* Keszitek egy strukturat*/
struct haromszog
	{
	int x,y,z;
	};

int main()
	{
	int i;
	FILE *fp;
	struct haromszog elem;

	fp=fopen("file.bin","wb");
	if (!fp)
		{
		return -1;
		}
	for (i = 0; i < 10; i++)
		{
		elem.x = i; elem.y = 2*i; elem.z = 3*i;
		fwrite(&elem, sizeof(struct haromszog), 1, fp);
		}
	fclose(fp);
	return 0;
	}
```

Ebben a példában definiáltam egy haromszog nevű struktúrát. Három int típusú szám alkotja. Ilyen számhármasokból írok a fájlba tízet.

Beolvasni sem bonyolultabb:

```
#include<stdio.h>

/* Keszitek egy strukturat*/
struct haromszog
	{
	int x,y,z;
	};

int main()
	{
	int i;
	FILE *fp;
	struct haromszog elem;

	fp=fopen("file.bin","rb");
	if (!fp)
		{
		return -1;
		}
	for (i = 0; i < 10; i++)
		{
		fread(&elem,sizeof(struct haromszog),1,fp);
		printf("%d\n",elem.x);
		}
	fclose(fp);
	return 0;
	}
```

A különbség a megnyitás módja „wb” és „rb” és az írás/olvasás függvény maga.

13.6 Pozicionálás a fájlokban

Lássuk, hogyan működik a pozicionálás az fseek függvénnyel.

```
#include<stdio.h>

struct haromszog
	{
	int x,y,z;
```

```
};

int main()
	{
	int index;
	FILE *fp;
	struct haromszog elem;

	fp=fopen("test.bin","rb");
	if (!fp)
		{
		return -1;
		}
	for (index = 9; index >= 0; index--)
		{
		fseek(fp, sizeof(struct haromszog)*index,SEEK_SET);
		fread(&elem,sizeof(struct haromszog), 1, fp);
		printf("%d\n",elem.x);
		}
	fclose(fp);
	return 0;
	}
```

A függvény deklarációja a következő

```
int fseek(FILE * fp, long int offset, int origin);
```

Az első paraméter a FILE pointer, a második a pozíció eltolásának mértéke. Ez az elem bájtokban mért nagyságának egész számú többszöröse. Az origin azaz az eltolás kezdete a következő makrókkal adható meg: SEEK_SET, SEEK_CUR és SEEK_END. Ezek jelentése: fájl elejétől, aktuális pozíciótól, illetve a fájl végétől számítjuk az offsetet, ami negatív is lehet.
A következő példában az fseek függvénnyel az utolsó rekordot keressük meg a fájlban, majd a rewind függvénnyel visszamegyünk a fájl elejére.

```
#include<stdio.h>

struct haromszog
	{
	int x,y,z;
	};

int main()
	{
	int index;
```

```
FILE *fp;
struct haromszog elem;

fp=fopen("test.bin","rb");
if (!fp)
      {
      return -1;
      }

fseek(fp, sizeof(struct haromszog), SEEK_END);
rewind(fp);

for (index=1; index <= 10; index++)
      {
      fread(&elem,sizeof(struct haromszog),1,fp);
      printf("%d\n",elem.x);
      }
fclose(fp);
return 0;
}
```

14 Programtervezési alapelvek

14.1 Strukturált programozás

Elmélet:
A strukturált programozás lényege a probléma részfeladatokra bontása és lépésenkénti megoldása, mely során kevés, de jól meghatározott vezérlési és adatszerkezeti elemet használunk. A három vezérlési szerkezet, amely felhasználásával az egyszerű utasításokból összetett utasítások, összetett utasításokból program egységek (blokkok, függvények) és program egységekből egy program modul felépül: a szekvencia, az elágazás és a ciklus.

14.1.1 Feladat:
Töltsünk fel egy tömböt egész számokkal, majd válogassuk szét az elemeket két csoportba: párosak és páratlanok. Írjuk meg a feladatot megoldó algoritmust egyetlen függvényben, illetve a strukturált programozási elv felhasználásával egyaránt. Vegyük észre, hogy a feladat egyetlen egységként kezelése csak kisebb problémák esetén eredményez átlátható programot!

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#define N 10                          //tömb mérete
```

```c
/* Nem strukturált megoldás */
int main() {
    int t[N];
    int paros[N], paratlan[N];
    int i=0, parosdb=0, paratlandb=0;
    srand(time(0));                 //véletlenszám generátor inicializálása
    for (i=0; i<N; i++)
            t[i]=rand()%100+1;      //tömbelemek [1,100] intervallumban
    for (i=0; i<N; i++) {           //szétválogatás
        if (t[i]%2) {
            paratlan[paratlandb]=t[i];
            paratlandb++;
        }
        else {
            paros[parosdb]=t[i];
            parosdb++;
        }
    }
    printf("\nParos elemek: ");
    for (i=0; i<parosdb; i++) printf("%d, ", paros[i]);
    printf("\nParatlan elemek: ");
    for (i=0; i<paratlandb; i++) printf("%d, ", paratlan[i]);
    printf("\n");
    return 0;
}
//---------------------------------------------------------------------------

/* Strukturált megoldás */
/* globális deklarációk */
struct ujtomb { int valogatas[N], db }; //szétválogatott tömb és a mérete
enum kriterium { PAROS, PARATLAN };     //válogatás kritériumai

//a tömb méretét megadó konstans makró minden fv-ben használható
void tombFeltolt(int * tomb);
void tombKiir(struct ujtomb t);
struct ujtomb tombKivalogat(int * tomb, int kriterium);

int main() {
    int t[N];
    tombFeltolt(t);
    struct ujtomb parostomb = tombKivalogat(t, PAROS);
    printf("\nParos elemek: ");
    tombKiir(parostomb);
    struct ujtomb paratlantomb = tombKivalogat(t, PARATLAN);
    printf("\nParatlan elemek: ");
```

```
    tombKiir(paratlantomb);
    return 0;
}

void tombFeltolt(int *tomb) {
    int i;
    srand(time(0));                  //véletlenszám generátor inicializálása
    for (i=0; i<N; i++)
            tomb[i]=rand()%100+1;    //értékadás tömbelemeknek
    return ;
}

void tombKiir(struct ujtomb t) {          //fv paramétere struktúra
    int i;
    for (i=0; i<t.db; i++)
        printf("%d, ", t.valogatas[i]);  //str. adattagra hivakozás op-a: .
    printf("\n");
    return ;
}

struct ujtomb tombKivalogat(int * tomb, int kriterium){
    struct ujtomb eredmeny;
    int i, j=0;
    for(i=0; i<N; i++) {
        if (tomb[i]%2==kriterium) {
            eredmeny.valogatas[j] = tomb[i];
            j++;
        }
    }
    eredmeny.db = j;
    return eredmeny;
}
```

14.2 Moduláris programozás

Elmélet:

Nagy programok fejlesztésekor a feladat bonyolultsága, a csapatmunka támogatása szükségessé teszi a probléma részekre bontását. Az egyes részek külön modulokat alkotnak, amelyeket a kész program összeállításakor illeszteni kell egymáshoz. Egy modul önálló, névvel rendelkező, önmagában értelmezhető fordítási egység. Önállóan tervezhető, kódolható, tesztelhető. Minden modulnál meg kell határozni a bemenő, kimenő paramétereket és azokat a függvényeket amelyeken keresztül a többi modullal kapcsolatot tart. Ezzel szemben az egy modulban megírt programot monolitikusnak nevezzük.

14.2.1 Feladat:

Vegyük az Alapalgoritmusok című fejezet 9.4.1 feladatát és írjuk át úgy, hogy megfeleljen a moduláris programozási alapelvnek: a két részfeladatot tegyük külön-külön modulba (.c kit. állományba). Ez egyszerűen megoldható, mert alapértelmezés szerint a függvények tárolási osztálya extern, ami a program összes moduljára kiterjedő láthatóságot biztosít.

```
/* main.c */
#include <stdio.h>
#include <time.h>
#include <stdlib.h>

// csak deklaráció
void a_feladat(int * tomb, int N, int minpont);
void b_feladat(int * tomb, int N, int maxpont);

// fv definíciók
void feltolt(int *tomb, int meret, int alsohatar, int felsohatar) {...}
void kiir(int tomb[], int meret) {...}
int main() {...}

/* a_feladat.c */
#include <stdio.h>

void kiir(int tomb[], int meret);

// fv definíciók
int keres(int *tomb, int meret, int k) {...}
void rendez(int *tomb, int meret) {...}
void a_feladat(int * tomb, int N, int minpont) {...}

/* b_feladat.c */
#include <stdlib.h>
#include <stdio.h>

// fv definíciók
int cmpfunc(const void * a, const void * b) {...}
void b_feladat(int * tomb, int N, int maxpont) {...}
```

14.3 Bottom-up (alulról felfelé) programtervezés

Elmélet:

Először a legelemibb részeket és azok megoldó algoritmusát készítjük el, majd ezek felhasználásával oldjuk meg az összetett feladatokat.

14.3.1 Feladat:

Kódoljuk az egydimenziós tömbökön elvégezhető alapvető műveleteket (elemek beolvasása, kiírása, automatikus tömbfeltöltés, összegzés, átlagolás, legnagyobb/legkisebb elem kiválasztása, keresés, rendezés, összekeverés) (lásd Alapalgoritmusok című fejezet). A megírt függvényeket gyűjtsük össze egy függvénykönyvtárba: azaz a definíciójukat tegyük külön modulba (tomb.c), a prototípusaikat pedig egy header állományba (tomb.h). Ezen könyvtári függvények hívásával oldjuk meg az alábbi feladatot.

Egy negyedéven keresztül hetente feljegyezzük a benzin árát. Az alábbi kérdésekre szeretnénk választ kapni:

1. Melyik volt a legmagasabb, ill. a legalacsonyabb ár az adott időszakban?
2. Mennyi volt az átlagos benzinár és az egyes heteken mennyivel tért el ettől a tényleges ár?
3. Monoton nőtt-e a benzin ára az adott időszakban?

```
/* tomb.c : egydimenziós tömbökön manipuláló függvények definíciója */
#include <time.h>
#include <stdlib.h>
#include <stdio.h>
#include "tomb.h"

...
/* ------------------------------------------------------------------ */

/* tomb.h : tomb.c függvényeinek deklarációját tartalmazó header fájl */
#ifndef TOMB_H_INCLUDED
#define TOMB_H_INCLUDED
extern void beolvas(int *tomb, int meret);
extern void feltolt(int *tomb, int meret, int alsoh, int felsoh);
extern void kiir(int tomb[], int meret);
extern void osszekever(int * tomb, int meret);
extern int osszegez(int tomb[], int meret);
extern double atlagol(int tomb[], int meret);
extern int minelem(int *tomb, int meret);
extern int maxelem(int *tomb, int meret);
extern int keres(int *tomb, int meret, int k); /* lineáris keresés */
extern void rendez(int *tomb, int meret);      /* minimumkiv. rendezés */
extern int cmpfunc(const void * a, const void * b);
extern void gyorsrendez(int * tomb, int meret);
int * binkeres(int *tomb, int meret, int k);
#endif // TOMB_H_INCLUDED
/* ------------------------------------------------------------------ */

/* main.c */
#include <stdio.h>
```

```
#include "tomb.h"

int main() {
    int N = 12;                    //egy negyedévben a hetek száma
    int tomb[N];
    int i, min, max;
    double atlag;
    beolvas(tomb, N);
    atlag = atlagol(tomb, N);
    min = minelem(tomb, N);
    max = maxelem(tomb, N);
    printf("\nLegkisebb elem: %d \nLegnagyobb elem: %d \nAtlag: %.2f\n",
       min, max, atlag);
    printf("Atlagtol valo elteresek:\n");
    for (i=0; i<N; i++)       // tömbelemek és átlagtól való eltérés kiírása
            printf("%d. \t %d fok \t %.2f\n", i, tomb[i], tomb[i]-atlag);

    //monotonitas vizsgalata
    for (i=1; i<N; i++)
        if (tomb[i-1]>tomb[i]) break;
    if (i==N) printf("A szamsorozat monoton novo.");
    else printf("A szamsorozat nem monoton novo.");
    return 0;
}
```

14.4 Top-down (felülről lefelé) programtervezés

Elmélet:

A problémamegoldás során a lépésenkénti finomítás elvét alkalmazzuk, azaz a feladat megoldását először átfogóan végezzük el. Ezután a feladatot részekre bontjuk, és a továbbiakban ezeket a részfeladatokat oldjuk meg. Így az egyes részeket egymástól függetlenül (de illesztve egymáshoz) írhatjuk, tesztelhetjük, javíthatjuk. Az egyes részeket tovább kell finomítani, amíg elemi részfeladatokig nem jutunk. Lásd Függvények című fejezet 8.2.2.1 feladat.

14.4.1 Feladat:

Keressük meg az összes N jegyű különös számot. N jegyű különös számnak nevezünk egy számot, ha számjegyeinek összege nagyobb, mint a nála N+1-el nagyobb szám számjegyeinek az összege. Például a 17 kétjegyű különös szám (N=2), mert számjegyeinek összege (8) nagyobb, mint a nála 3-al (N+1) nagyobb szám (20) számjegyeinek az összege (2). A feladatot a top-down tervezési elvet követve végezzük el.

1. lépés: Írjuk meg azt a függvényt, amely megoldja a feladatot, azaz megállapítja egy számról, hogy különös-e (a definíció szerint). Ha igen, térjen vissza 1-el; egyébként 0-val.

```
int kulonosszam_e(int szam, int N) {
    if (szamjegyek_osszege(szam) > szamjegyek_osszege(szam+N+1))
        return 1;
    else return 0;
}
```

2. lépés: A függvény írásakor egy megoldandó részfeladattal szembesülünk. Meg kell határozni a vizsgált számok számjegyeinek összegét.

```
int szamjegyek_osszege(int n) {
    int osszeg=0;
    while (n>0) {
        osszeg = osszeg+(n%10);
        n = n/10;
    }
    return osszeg;
}
```

3. lépés: A probléma tovább nem bontható. Írjuk meg a main függvényt, állítsuk össze a programot.

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

int main() {
      int N, i;
      printf("Hany jegyu kulonos szamokat keressek? ");
      scanf("%d", &N);
      for (i=pow(10.0, (double)(N-1)); i<pow(10.0, (double)N); i++){
            if (kulonosszam_e(i, N)==1)
                  printf("%d, ", i);
      }
      return 0;
}
```

15 Irodalom

1. B. Kernighan, D. M. Ritchie: The C Programming Language (ANSI C) (2nd edition), Prentice Hall Software Series
2. Donald Knuth: A számítógép-programozás művészete
3. Baksáné V. Erika: Sorozatok nevezetes algoritmusai – példatár (2014)

16 Köszönetnyilvánítás

Az oktatási anyag kidolgozása a Miskolci Egyetem társadalmi – gazdasági szerepének fejlesztése, különös tekintettel a duális képzési típusú megoldásokra témakörű K+F projektje keretében - TÁMOP-4.1.1.F-13/1-2013-0010 - az „ELTÉRŐ UTAK A SIKERES ÉLETHEZ” projekt részeként – az Új Széchenyi Terv keretében – az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósult meg.